Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

A Multicore Tool for Constraint Solving

Roberto Amadini, Maurizio Gabbrielli, Jacopo Mauro
Department of Computer Science and Engineering, University of Bologna / Lab. Focus INRIA
{amadini,gabbri,jmauro}@cs.unibo.it

Abstract

In Constraint Programming (CP), a portfolio solver uses a variety of different solvers for solving a given Constraint Satisfaction / Optimization Problem. In this paper we introduce `sunny-cp2`: the first parallel CP portfolio solver that enables a dynamic, cooperative, and simultaneous execution of its solvers in a multicore setting. It incorporates state-of-the-art solvers, providing also a usable and configurable framework. Empirical results are very promising. `sunny-cp2` can even outperform the performance of the oracle solver which always selects the best solver of the portfolio for a given problem.

1 Introduction

The *Constraint Programming* (CP) paradigm enables to express complex relations in form of constraints to be satisfied. CP allows to model and solve *Constraint Satisfaction Problems* (CSPs) as well as *Constraint Optimization Problems* (COPs) [Rossi *et al.*, 2006]. Solving a CSP means finding a solution that satisfies all the constraints of the problem, while a COP is a generalized CSP where the goal is to find a solution that minimises or maximises an objective function.

A fairly recent trend in CP is solving a problem by means of portfolio approaches [Gomes and Selman, 2001], which can be seen as instances of the more general Algorithm Selection problem [Rice, 1976]. A *portfolio solver* is a particular solver that exploits a collection of $m > 1$ constituent solvers $s_1, \dots, s_m$ to get a globally better solver. When a new, unseen problem p comes, the portfolio solver tries to predict the best constituent solver(s) $s_{i_1}, \dots, s_{i_k}$ (with $1 \leq k \leq m$) for solving p and then runs them.

Unfortunately —despite their proven effectiveness— portfolio solvers are scarcely used outside the walls of solvers competitions, and often confined to SAT solving. Regarding the CP field, the first portfolio solver (for solving CSPs only) was CPHydra [O'Mahony *et al.*, 2008] that in 2008 won the International CSP Solver Competition. In 2014, the sequential portfolio solver `sunny-cp` [Amadini *et al.*, 2015] attended the *MiniZinc Challenge* (MZC) [Stuckey *et al.*, 2010] with respectable results (4th out of 18). To the best of our knowledge, no similar tool exist for solving both CSPs and COPs.

In this paper we take a major step forward by introducing `sunny-cp2`, a significant enhancement of `sunny-cp`. Improvements are manifold. Firstly, `sunny-cp2` is *parallel*: it exploits multicore architectures for (possibly) running more constituent solvers simultaneously. Moreover, while most of the parallel portfolio solvers are static (i.e., they decide off-line the solvers to run, regardless of the problem to be solved), `sunny-cp2` is instead *dynamic:* the solvers scheduling is predicted on-line according to a generalization of the SUNNY algorithm [Amadini *et al.*, 2014c]. Furthermore, for COPs, the parallel execution is also *cooperative*: a running solver can exploit the current best bound found by another one through a configurable *restarting* mechanism. Finally, `sunny-cp2` enriches `sunny-cp` by incorporating new, state-of-the-art solvers and by providing a framework which is more *usable* and *configurable* for the end user.

We validated `sunny-cp2` on two benchmarks of CSPs and COPs (about 5000 instances each). Empirical evidences show very promising results: its performance is very close (and sometimes even better) to that of the *Virtual Best Solver* (VBS), i.e., the oracle solver which always selects the best solver of the portfolio for a given problem. Moreover, for COPs, `sunny-cp2` using $c = 1, 2, 4, 8$ cores always outperforms the *Virtual c-Parallel Solver* (VPS$_c$), i.e., a static portfolio solver that runs in parallel a fixed selection of the best c solvers of the portfolio.

Paper Structure. In Section 2 we generalize the SUNNY algorithm for a multicore setting. In Section 3 we describe the architecture and the novelties of `sunny-cp2`, while in Section 4 we discuss its empirical evaluation. In Section 5 we report the related work before concluding in Section 6.

2 Generalizing the SUNNY Algorithm

SUNNY is the algorithm that underpins `sunny-cp`. Fixed a solving timeout T and a portfolio Π, SUNNY exploits instances similarity to produce a sequential schedule $\sigma = [(s_1, t_1), \dots, (s_n, t_n)]$ where solver s_i has to be run for t_i seconds and $\sum_{i=1}^{n} t_i = T$. For any input problem p, SUNNY uses a *k-Nearest Neighbours* (k-NN) algorithm to select from a training set of known instances the subset $N(p, k)$ of the k instances closer to p. According to the $N(p, k)$ instances, SUNNY relies on three heuristics: h_{sel}, for *selecting* the most promising solvers to run; h_{all}, for *allocating* to each solver a certain runtime (the more a solver is promising, the more time is allocated); and h_{sch}, for *scheduling* the sequential execution of the solvers according to their presumed speed. These heuristics depend on the application domain. For example, for CSPs h_{sel} selects the smallest sub-portfolio $S \subseteq \Pi$ that solves the most instances in $N(p,k)$, by using the solving time for breaking ties. h_{all} allocates to each $s_i \in S$ a time t_i proportional to the instances that S can solve in $N(p,k)$, while h_{sch} sorts the solvers by increasing solving time in $N(p,k)$. For COPs the approach is analogous, but different performance metrics are used [Amadini *et al.*, 2014b].

As an example, let p be a CSP, $\Pi = \{s_1, s_2, s_3, s_4\}$ a portfolio of 4 solvers, $T = 1800$ seconds the solving timeout, $N(p,k) = \{p_1, ..., p_4\}$ the $k = 4$ neighbours of p, and the runtimes of solver s_i on problem p_j defined as in Table 1. In this case, the smallest sub-portfolios that solve the most instances are $\{s_1, s_2, s_3\}$, $\{s_1, s_2, s_4\}$, and $\{s_2, s_3, s_4\}$. The heuristic h_{sel} selects $S = \{s_1, s_2, s_4\}$ because these solvers are faster in solving the instances in the neighbourhood. Since s_1 and s_4 solve 2 instances and s_2 solves 1 instance, h_{all} splits the solving time window $[0, T]$ into 5 slots: 2 allocated to s_1 and s_4, and 1 to s_2. Finally, h_{sch} sorts the solvers by increasing solving time. The final schedule produced by SUNNY is therefore $\sigma = [(s_4, 720), (s_1, 720), (s_2, 360)]$.

	p_1	p_2	p_3	p_4
s_1	T	**3**	T	**278**
s_2	**593**	T	T	T
s_3	T	**36**	**1452**	T
s_4	T	T	**122**	**60**

Table 1: Runtimes (in seconds). T means the solver timeout.

For a detailed description and evaluation of SUNNY we refer the reader to [Amadini *et al.*, 2014c; 2014b]. Here we focus instead on how we generalized SUNNY for a multicore setting where $c \geq 1$ cores are available and, typically, we have more solvers than cores. `sunny-cp2` uses a *dynamic* approach: the schedule is decided on-line according to the instance to be solved. The approach used is rather simple: starting from the sequential schedule σ produced by SUNNY on a given problem, we first detect the $c-1$ solvers of σ having the largest allocated time, and we assign a different core to each of them. The other solvers of σ (if any) are scheduled on the last core, by preserving their original order in σ and by widening their allocated times to cover the entire time window $[0, T]$.

Formally, given c cores and the sequential schedule $\sigma = [(s_1, t_1), \dots, (s_n, t_n)]$ computed by SUNNY, we define a total order $\prec_\sigma$ such that $s_i \prec_\sigma s_j \Leftrightarrow t_i > t_j \vee (t_i = t_j \wedge i < j)$, and a ranking function $rank_\sigma$ such that $rank_\sigma(s) = i$ if and only if s is the i-th solver of σ according to $\prec_\sigma$. We define the *parallelisation* of σ on c cores as a function $\mathbb{P}_{\sigma,c}$ which associates to each core $i \in \{1, \dots, c\}$ a sequential schedule $\mathbb{P}_{\sigma,c}(i)$ such that:

$$\mathbb{P}_{\sigma,c}(i) = \begin{cases} [\,] & \text{if } i > n \\ [(s,T)] & \text{if } i = rank_\sigma(s) \wedge 1 \leq i < c \\ [(s, \frac{T}{T_c}t) \mid (s,t) \in \sigma \wedge rank_\sigma(s) \geq c] & \text{if } i = c \end{cases}$$

where $T_c = \sum\{t \mid (s,t) \in \sigma \wedge rank_\sigma(s) \geq c\}$.

Let us consider the schedule of the previous example: $\sigma = [(s_4, 720), (s_1, 720), (s_2, 360)]$. With $c = 2$ cores, the parallelisation of σ is defined as $\mathbb{P}_{\sigma,2}(1) = [(s_4, 1800)]$ and $\mathbb{P}_{\sigma,2}(2) = [(s_1, 1800/1080 \times 720), (s_2, 1800/1080 \times 360)]$.

The rationale behind the schedule parallelisation $\mathbb{P}_{\sigma,c}$ is that SUNNY aims to allocate more time to the most promising solvers, scheduling them according to their speed. Assuming a good starting schedule σ, its parallelisation $\mathbb{P}_{\sigma,c}$ is rather straightforward and it is easy to prove that, assuming an independent execution of the solvers without synchronization and memory contention issues, $\mathbb{P}_{\sigma,c}$ can never solve less problems than σ.

Obviously, different ways to parallelise the sequential schedule are possible. Here we focus on what we think to be one of the most simple yet promising ones. An empirical comparison of other parallelisation methods is outside the scope of this paper.

3 SUNNY-CP 2

`sunny-cp2` solves both CSPs and COPs encoded in the *MiniZinc* language [Nethercote *et al.*, 2007], nowadays the de-facto standard for modelling CP problems. By default, `sunny-cp2` uses a portfolio Π of 12 solvers disparate in their nature. In addition to the 8 solvers of `sunny-cp` (viz., Chuffed, CPX, G12/CBC, G12/FD, G12/LazyFD, G12/Gurobi, Gecode, and MinisatID) it includes new, state-of-the-art solvers able to win four gold medals in the MZC 2014, namely: Choco, HaifaCSP, iZplus, and OR-Tools.

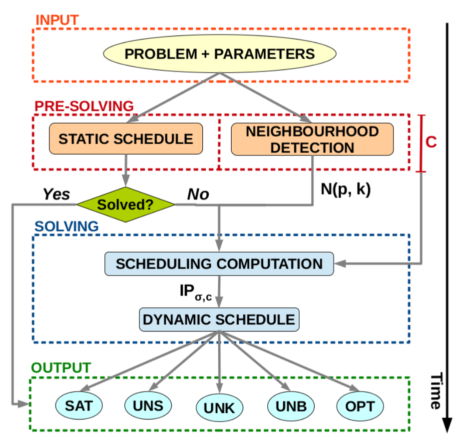

Figure 1: `sunny-cp2` architecture.

Figure 1 summarizes the execution flow of `sunny-cp2`. It takes as input a problem p to be solved and a list of input parameters specified by the user (e.g., the timeout T used by SUNNY, the number of cores c to use, etc.). If a parameter is not specified, a corresponding default value is used.

The solving process then relies on two sequential steps, later detailed: *(i)* the *pre-solving* phase, where a static schedule of solvers may be run and the instance neighbourhood is computed; *(ii)* the *solving* phase, which runs the dynamic schedule on different cores.

If p is a CSP, the output of `sunny-cp2` can be either *SAT* (a solution exists for p), *UNS* (p has no solutions), or *UNK* (`sunny-cp2` can not say anything about p). In addition, if p is a COP, two more answers are possible: *OPT* (`sunny-cp2` proves the optimality of a solution), or *UNB* (p is unbounded).

3.1 Pre-solving

The pre-solving step consists in the simultaneous execution on c cores of two distinct tasks: the execution of a (possibly empty) static schedule, and the neighbourhood detection.

Running for a short time a static schedule has proven to be effective. It enables to solve those instances that some solvers can solve very quickly (e.g., see [Kadioglu *et al.*, 2011]) and, for COPs, to quickly find good sub-optimal solutions [Amadini and Stuckey, 2014].

The *static schedule* $\overline{\sigma} = [(s_1, t_1), \ldots, (s_n, t_n)]$ is a prefixed schedule of $n \geq 0$ solvers decided off-line, regardless of the input problem p. To execute it, `sunny-cp2` applies a "First-Come, First-Served" policy by following the schedule order. Formally, the first m solvers $s_1, \ldots, s_m$ with $m = \min(c, n)$ are launched on different cores. Then, for $i = m + 1, ..., n$, the solver s_i is started as soon as a solver s_j (with $1 \leq j \leq i$) terminates its execution without solving p. If a solver s_i fails prematurely before t_i (e.g., due to memory overflows or unsupported constraints) then s_i is discarded from the portfolio for avoiding to run it again later. If instead s_i reached its timeout, then it is just suspended: if it has to run again later, it will be resumed instead of restarted from scratch. The user has the possibility of setting the static schedule as an input parameter. For simplicity, the static schedule of `sunny-cp2` is empty by default.

When solving COPs, the timeout defined by the user may be overruled by `sunny-cp2` that examines the solvers behaviour and may decide to delay the interruption of a solver. Indeed, a significant novelty of `sunny-cp2` is the use of a *waiting threshold* T_w. A solver s scheduled for t seconds is never interrupted if it has found a new solution in the last T_w seconds, regardless of whether the timeout t is expired or not. The underlying logic of T_w is to not stop a solver which is actively producing solutions. Thus, if $T_w > 0$ a solver may be executed for more than its allocated time. Setting T_w to a high value might therefore delay, and thus hinder, the execution of the other solvers.

For COPs, `sunny-cp2` also enables the *bounds communication* between solvers: the sub-optimal solutions found by a solver are used to narrow the search space of the other scheduled solvers. In [Amadini and Stuckey, 2014] this technique is successfully applied in a sequential setting, where the best objective bound found by a solver is exploited by the next scheduled ones. Things get more complicated in a multicore setting, where solvers are run simultaneously as black boxes and there is no support for communicating bounds to another running solver without interrupting its solving process. We decided to overcome this problem by using a *restarting threshold* T_r. If a running solver s finds no solution in the last T_r seconds, and its current best bound v is obsolete w.r.t. the overall best bound v' found by another solver, then s is restarted with the new bound v'. The reason behind T_r is that restarting an active solver is risky and potentially harmful even when its current best bound is actually obsolete. However, also ignoring the solutions found by other solvers could mean to neglect valuable information. The choice of T_r is hence critical: a too small value might cause too many restarts, while a big threshold inhibits the bound communication. Care should be taken also because restarting a solver means to lose all the knowledge it gained during the search.

After performing some empirical investigations, we found reasonable to set the default values of T_w and T_r to 2 and 5 seconds respectively. The user can however configure such parameters as she wishes, even by defining a customized setting for each different solver of the portfolio.

The *neighbourhood detection* phase begins in parallel with the static schedule $\overline{\sigma}$ execution only after all the solvers of $\overline{\sigma}$ has been started. This phase has lower priority than $\overline{\sigma}$ since its purpose is not quickly finding solutions, but detecting the neighbourhood $N(p, k)$ later on used to compute the dynamic schedule. For this reason, the computation of the neighbourhood starts as soon as one core is free (i.e., the number of running solvers is smaller than c). The first step of pre-scheduling is the extraction of the *feature vector* of p, i.e., a collection of numerical attributes (e.g., number of variables, of constraints, etc.) that characterizes the instance p. The feature vector is then used for detecting the set $N(p, k)$ of the k nearest neighbours of p within a dataset of known instances.

The default dataset Δ of `sunny-cp2` is the union of a set Δ_{CSP} of 5527 CSPs and a set Δ_{COP} of 4988 COPs, retrieved from the instances of the MiniZinc 1.6 benchmarks, the MZCs 2012–2014, and the International CSP Solver Competitions 2008/09. `sunny-cp2` provides a default knowledge base that associates to every instance $p_i \in \Delta$ its feature vector and the runtime information of each solver of the portfolio on p_i (e.g., solving time, anytime performance, etc.). In particular, the feature vectors are used for computing $N(p, k)$ while the runtime information are later used for computing the dynamic schedule σ by means of SUNNY algorithm.

The feature extractor used by `sunny-cp2` is the same as `sunny-cp`.[1] The neighbourhood size is set by default to $k = 70$: following [Duda *et al.*, 2000] we chose a value of k close to $\sqrt{n}$, where n is the number of training samples. Note that during the pre-solving phase only $N(p, k)$ is computed. This is because SUNNY requires a timeout T for computing the schedule σ. The total time taken by the pre-solving phase (C in Figure 1) must therefore be subtracted from the initial timeout T (by default, $T = 1800$ seconds as in `sunny-cp`). This can be done only when the pre-solving ends, since C is not predictable in advance.

The pre-solving phase terminates when either: *(i)* a solver of the static schedule $\overline{\sigma}$ solves p, or *(ii)* the neighbourhood

[1] The feature extractor `mzn2feat` is available at https://github.com/CP-Unibo/mzn2feat. For further information please see [Amadini *et al.*, 2014a] and [Amadini *et al.*, 2015].

computation and all the solvers of $\overline{\sigma}$ have finished their execution. In the first case `sunny-cp2` outputs the solving outcome and stops its execution, skipping the solving phase since the instance is already solved.

3.2 Solving

The solving phase receives in input the time C taken by presolving and the neighbourhood $N(p, k)$. These parameters are used by SUNNY to dynamically compute the sequential schedule σ with a reduced timeout $T - C$. Then, σ is parallelised according to the $\mathbb{P}_{\sigma,c}$ operator defined in Section 2. Once computed $\mathbb{P}_{\sigma,c}$, each scheduled solver is run on p according to the input parameters. If a solver of $\mathbb{P}_{\sigma,c}$ had already been run in the static schedule $\overline{\sigma}$, then its execution is resumed instead of restarted. If p is a COP, the solvers execution follows the approach explained in Section 3.1 according to T_w and T_r thresholds. As soon as a solver of σ solves p, the execution of `sunny-cp2` is interrupted and the solving outcome is outputted.

Note that we decided to make `sunny-cp2` an *anytime solver*, i.e., a solver that can run indefinitely until a solution is found. For this reason we let the solvers run indefinitely even after the timeout T until a solution is found. Moreover, at any time we try to have exactly c running solvers. In particular, if c is greater or equal than the portfolio size no prediction is needed and we simply run a solver per core for indefinite time. When a scheduled solver fails during its execution, the overall solving process is not affected since `sunny-cp2` will simply run the next scheduled solver, if any, or another one in its portfolio otherwise.

Apart from the scheduling parallelisation, `sunny-cp2` definitely improves `sunny-cp` also from the engineering point of view. One of the main purposes of `sunny-cp2` is indeed to provide a framework which is easy to use and configure by the end user. First of all, `sunny-cp2` provides utilities and procedures for adding new constituent solvers to Π and for customizing their settings. Even though the feature extractor of `sunny-cp2` is the same as `sunny-cp`, the user can now define and use her own tool for feature processing. The user can also define her own knowledge base, starting from raw comma-separated value files, by means of suitable utility scripts. Furthermore, `sunny-cp2` is much more parametrisable than `sunny-cp`. Apart from the aforementioned T_w and T_r parameters, `sunny-cp2` provides many more options allowing, e.g., to set the number c of cores to use, limit the memory usage, ignore the search annotations, and specify customized options for each constituent solver.

The source code of `sunny-cp2` is entirely written in Python and publicly available at https://github.com/CP-Unibo/sunny-cp.

4 Validation

We validated the performance of `sunny-cp2` by running it with its default settings on every instance of Δ_{CSP} (5527 CSPs) and Δ_{COP} (4988 COPs). Following the standard practices, we used a 10-fold cross-validation: we partitioned each dataset in 10 disjoint folds, treating in turn one fold as test set and the union of the remaining folds as the training set.

Fixed a solving timeout T, different metrics can be adopted to measure the effectiveness of a solver s on a given problem p. If p is a CSP we typically evaluate whether, and how quickly, s can solve p. For this reason we use two metrics, namely `proven` and `time`, to measure the solved instances and the solving time. Formally, if s solves p in $t < T$ seconds, then $\texttt{proven}(s, p) = 1$ and $\texttt{time}(s, p) = t$; otherwise, $\texttt{proven}(s, p) = 0$ and $\texttt{time}(s, p) = T$. A straightforward generalization of these two metrics for COPs can be obtained by setting $\texttt{proven}(s, p) = 1$ and $\texttt{time}(s, p) = t$ if s proves in $t < T$ seconds the optimality of a solution for p, the unsatisfiability of p or its unboundedness. Otherwise, $\texttt{proven}(s, p) = 0$ and $\texttt{time}(s, p) = T$. However, a major drawback of such a generalization is that the solution quality is not addressed. To overcome this limitation, we use in addition the `score` and `area` metrics introduced in [Amadini and Stuckey, 2014].

The `score` metric measures the quality of the best solution found by a solver s at the stroke of the timeout T. More specifically, $\texttt{score}(s, p)$ is a value in $[0.25, 0.75]$ linearly proportional to the distance between the best solution that s finds and the best known solution for p. An additional reward ($\texttt{score} = 1$) is given if s is able to prove optimality, while a punishment ($\texttt{score} = 0$) is given if s does not provide any answer. Differently from `score`, the `area` metric evaluates the *behaviour* of a solver in the whole solving time window $[0, T]$ and not only at the time edge T. It considers the area under the curve that associates to each time instant $t_i \in [0, T]$ the best objective value v_i found by s in t_i seconds. The `area` value is properly scaled in the range $[0, T]$ so that the more a solver is slow in finding good solutions, the more its `area` is high (in particular, $\texttt{area} = T$ if and only if $\texttt{score} = 0$). The `area` metric folds in a number of measures the quality of the best solution found, how quickly any solution is found, whether optimality is proven, and how quickly good solutions are found. For a formal definition of `area` and `score` we refer the reader to [Amadini and Stuckey, 2014].

We ran `sunny-cp2` on all the instances of Δ_{CSP} and Δ_{COP} within a timeout of $T = 1800$ seconds by varying the number c of cores in $\{1, 2, 4, 8\}$. We compared `sunny-cp2` against the very same version of `sunny-cp` submitted to MZC 2014[2] and the Virtual Best Solver (VBS), i.e., the oracle portfolio solver that —for a given problem and performance metric— always selects the best solver to run. Moreover, for each $c \in \{1, 2, 4, 8\}$ we consider as additional baseline the *Virtual c-Parallel Solver* (VPS_c), i.e., a static portfolio solver that always runs in parallel a fixed selection of the best c solvers of the portfolio Π. The VPS_c is a static portfolio since its solvers are selected in advance, regardless of the problem p to be solved. Obviously, the portfolio composition of VPS_c depends on a given evaluation metric. For each $\mu \in \{\texttt{proven}, \texttt{time}, \texttt{score}, \texttt{area}\}$ we therefore defined a corresponding VPS_c by selecting the best c solvers of Π according to the average value of μ over the Δ_{CSP} or Δ_{COP} instances. Note that VPS_c is an "ideal" solver, since

[2] The source code of `sunny-cp` is publicly available at https://github.com/CP-Unibo/sunny-cp/tree/mznc14

Metric	sunny-cp	sunny-cp2				VPS				VBS
		1 core	2 cores	4 cores	8 cores	1 core	2 cores	4 cores	8 cores	
proven (%)	83.26	95.26	99.11	**99.38**	99.35	89.04	93.51	95.19	99.24	**100**
time (s)	504.10	223.21	136.04	112.60	**112.32**	297.30	211.13	176.81	**68.52**	**54.30**

Table 2: Experimental results over CSPs.

Metric	sunny-cp	sunny-cp2				VPS				VBS
		1 core	2 cores	4 cores	8 cores	1 core	2 cores	4 cores	8 cores	
proven (%)	71.55	74.40	74.94	75.68	**76.34**	61.33	63.45	65.60	75.86	**76.30**
time (s)	594.79	501.35	482.95	469.74	**454.54**	718.86	682.35	645.68	463.63	**457.00**
score $\times$ 100	90.50	92.26	93.00	93.45	**93.62**	82.80	85.99	89.53	90.79	**93.63**
area (s)	257.86	197.44	149.33	138.94	**130.53**	314.07	266.11	188.20	178.81	**132.28**

Table 3: Experimental results over COPs.

it is an *upper bound* of the real performance achievable by actually running in parallel all its solvers. An approach similar to VPS_c has been successfully used by the SAT portfolio solver ppfolio [Roussel, 2011]. The VPS_1 is sometimes also called *Single Best Solver* (SBS) while $\text{VPS}_{|\Pi|}$ is exactly equivalent to the *Virtual Best Solver* (VBS).

In the following, we will indicate with $\texttt{sunny-cp2}_{[c]}$ the version of sunny-cp2 exploiting c cores. Empirical results on CSPs and COPs are summarized in Table 2 and 3 reporting the average values of all the aforementioned evaluation metrics. On average, sunny-cp2 remarkably outperforms all its constituent solvers as well as sunny-cp for both CSPs and COPs. Concerning CSPs, a major boost is given by the introduction of HaifaCSP solver in the portfolio. HaifaCSP solves almost 90% of Δ_{CSP} problems. However, as can be seen from the Tables 2 and 3, $\texttt{sunny-cp2}_{[1]}$ solves more than 95% of the instances and, for $c > 1$, $\texttt{sunny-cp2}_{[c]}$ solves nearly all the problems. This means that just few cores are enough to almost reach the VBS performance. The best proven performance is achieved by $\texttt{sunny-cp2}_{[4]}$.[3] Nevertheless, the average time of $\texttt{sunny-cp2}_{[8]}$ is slightly slower. However, difference are minimal: $\texttt{sunny-cp2}_{[4]}$ and $\texttt{sunny-cp2}_{[8]}$ are virtually equivalent. Note that $\texttt{sunny-cp2}_{[c]}$ is always better than VPS_c in terms of proven and, for $c < 8$, also in terms of time. Furthermore, $\texttt{sunny-cp2}_{[1]}$ solves more instances than VPS_4. In other words, running sequentially the solvers of sunny-cp2 on a single core allows to solve more instances than running independently the four solvers of the portfolio that solve the most number of problems of Δ_{CSP}. Analogously, $\texttt{sunny-cp2}_{[2]}$ solves more instances than VPS_4. This witnesses that in a multicore setting, where there are typically more available solvers than cores, sunny-cp2 can be very effective.

Even better results are reached by sunny-cp2 on the instances of Δ_{COP}, where there is not a clear dominant solver like HaifaCSP for Δ_{CSP}. sunny-cp2 outperforms sunny-cp in all the considered metrics, and the gain of performance in this case is not only due to the introduction of new solvers. Indeed, the best solver according to time and proven is Chuffed, which was already present in sunny-cp. For each $c \in \{1, 2, 4, 8\}$ and for each performance metric, $\texttt{sunny-cp2}_{[c]}$ is always better than the corresponding VPS_c. Moreover, as for CSPs, sunny-cp2 has very good performances even with few cores. For example, by considering the proven, time, and score results, $\texttt{sunny-cp2}_{[1]}$ is always better than VPS_4 and its score even outperforms that of VPS_8. This clearly indicates that the dynamic selection of solvers, together with the bounds communication and the waiting/restarting policies implemented by sunny-cp2, makes a remarkable performance difference.

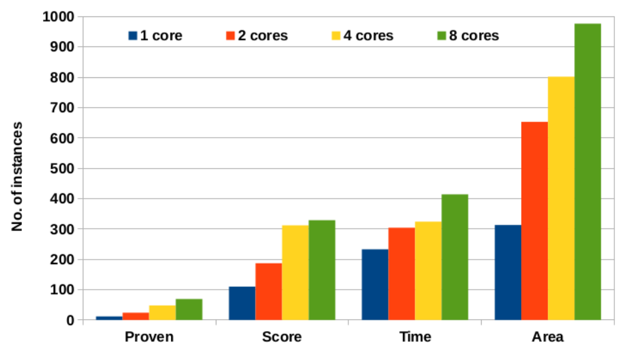

Figure 2: Number of COPs for which sunny-cp2 outperforms the VBS.

Results also show a nice monotonicity: for all the considered metrics, if $c > c'$ then $\texttt{sunny-cp2}_{[c]}$ is better than $\texttt{sunny-cp2}_{[c']}$. In particular, it is somehow impressive to see that on average sunny-cp2 is able to *outperform* the VBS in terms of time, proven, and area, and that for score the performance difference is negligible (0.01%). This is better shown in Figure 2 depicting for each performance metric the number of times that sunny-cp2 is able to overcome the performance of the VBS with $c = 1, 2, 4, 8$ cores. As c increases, the number of times the VBS is beaten gets bigger. In particular, the time and area results show that sunny-cp2 can be very effective to reduce

[3] The practically negligible difference with $\texttt{sunny-cp2}_{[8]}$ is probably due to synchronization and memory contention issues (e.g., cache misses) that slow down the system when 8 cores are exploited.

	Choco	Chuffed	CPX	G12/FD	Gecode	OR-Tools	Other solvers	`sunny-cp2`
t_{best} (s)	4.05	52.49	**2.42**	2.87	3.31	1.46	N/A	5.00
v_{best}	959	**958**	**958**	959	959	959	N/A	**958**
`time`	T	T	T	T	T	T	T	**10.56**

Table 4: Benefits of `sunny-cp2` on a RCPSP instance. $T = 1800$ indicates the solvers timeout.

the solving time both for completing the search and, especially, for quickly finding sub-optimal solutions. For example, for 414 problems `sunny-cp2`$_{[8]}$ has a lower `time` than VBS, while for 977 instances it has a lower `area`. Table 4 reports a clear example of the `sunny-cp2` potential on an instance of the Resource-Constrained Project Scheduling Problem (RCPSP) [Brucker *et al.*, 1999] taken from the MiniZinc-1.6 suite.[4] Firstly, note that just half of the solvers of Π can find at least a solution. Second, these solvers find their best bound v_{best} very quickly (i.e, in a time t_{best} of between 1.46 and 4.05 seconds) but none of them is able to prove the optimality of the best bound $v^* = 958$ within a timeout of $T = 1800$ seconds. Conversely, `sunny-cp2` finds v^* and proves its optimality in less than 11 seconds. Exploiting the fact that CPX finds v^* in a short time (for CPX $t_{\text{best}} = 2.42$ seconds, while `sunny-cp2` takes 5 seconds due to the neighbourhood detection and scheduling computation), after $T_r = 5$ seconds any other scheduled solver is restarted with the new bound v^*. Now, Gecode can prove almost instantaneously that v^* is optimal, while without this help it can not even find it in half an hour of computation.

5 Related Work

The parallelisation of CP solvers does not appear as fruitful as for SAT solvers where techniques like clause sharing are used. As an example, in the MZC 2014 the possibility of multiprocessing did not lead to remarkable performance gains, despite the availability of eight logical cores (in a case, a parallel solver was even significantly worse than its sequential version).

Portfolio solvers have proven their effectiveness in many international solving competitions. For instance, the SAT portfolio solvers 3S [Kadioglu *et al.*, 2011] and CSHC [Malitsky *et al.*, 2013] won gold medals in SAT Competition 2011 and 2013 respectively. SATZilla [Xu *et al.*, 2008] won the SAT Challenge 2012, CPHydra [O'Mahony *et al.*, 2008] the Constraint Solver Competition 2008, the ASP portfolio solver claspfolio [Hoos *et al.*, 2014] was gold medallist in different tracks of the ASP Competition 2009 and 2011, ArvandHerd [Valenzano *et al.*, 2012] and IBaCoP [Cenamor *et al.*, 2014] won some tracks in the Planning Competition 2014.

Surprisingly enough, only a few portfolio solvers are parallel and even fewer are the dynamic ones selecting on-line the solvers to run. We are aware of only two dynamic and parallel portfolio solvers that attended a solving competition, namely p3S [Malitsky *et al.*, 2012] (in the SAT Challenge 2012) and IBaCoP2 [Cenamor *et al.*, 2014] (in the Planning Competition 2014). Unfortunately, a comparison of `sunny-cp2` with these tools is not possible because their source code is not publicly available and they do not deal with COPs.

Apart from the aforementioned CPHydra and `sunny-cp` solvers, another portfolio approach for CSPs is Proteus [Hurley *et al.*, 2014]. However, with the exception of a preliminary investigation about a CPHydra parallelisation [Yun and Epstein, 2012], all these solvers are sequential and except for `sunny-cp` they solve just CSPs. Hence, to the best of our knowledge, `sunny-cp2` is today the only parallel and dynamic CP portfolio solver able to deal with also optimization problems.

The parallelisation of portfolio solvers is a hot topic which is drawing some attention in the community. For instance, parallel extensions of well-known sequential portfolio approaches are studied in [Hoos *et al.*, 2015b]. In [Hoos *et al.*, 2015a] ASP techniques are used for computing a static schedule of solvers which can even be executed in parallel, while [Cire *et al.*, 2014] considers the problem of parallelising restarted backtrack search for CSPs.

6 Conclusions

In this paper we introduced `sunny-cp2`, the first parallel CP portfolio solver able to dynamically schedule the solvers to run and to solve both CSPs and COPs encoded in the MiniZinc language. It incorporates state-of-the-art solvers, providing also a usable and configurable framework.

The performance of `sunny-cp2`, validated on heterogeneous and large benchmarks, is promising. Indeed, `sunny-cp2` greatly outperforms all its constituent solvers and its earlier version `sunny-cp`. It can be far better than a ppfolio-like approach [Roussel, 2011] which statically determine a fixed selection of the best solvers to run. For CSPs, `sunny-cp2` almost reaches the performance of the Virtual Best Solver, while for COPs `sunny-cp2` is even able to outperform it.

We hope that `sunny-cp2` can stimulate the adoption and the dissemination of CP portfolio solvers. Indeed, `sunny-cp` was the only portfolio entrant of MiniZinc Challenge 2014. We are interested in submitting `sunny-cp2` to the 2015 edition in order to compare it with other possibly parallel portfolio solvers.

There are many lines of research that can be explored, both from the scientific and engineering perspective. As a future work we would like to extend the `sunny-cp2` by adding new, possibly parallel solvers. Moreover, different parallelisations, distance metrics, and neighbourhood sizes can be evaluated.

Given the variety of parameters provided by `sunny-cp2`, it could be also interesting to exploit Algorithm Configuration techniques [Hutter *et al.*, 2011; Kadioglu *et al.*, 2010] for the automatic tuning of the `sunny-cp2` parameters, as well as the parameters of its constituent solvers.

[4] The model is `rcpsp.mzn` while the data is in `la10_x2.dzn`

Finally, we are also interested in making `sunny-cp2` more usable and portable, e.g., by pre-installing it on virtual machines or multi-container technologies.

Acknowledgements

We would like to thank the staff of the Optimization Research Group of NICTA (National ICT of Australia) for allowing us to use Chuffed and G12/Gurobi solvers, as well as for granting us the computational resources needed for building and testing `sunny-cp2`. Thanks also to Michael Veksler for giving us an updated version of the HaifaCSP solver.

References

[Amadini and Stuckey, 2014] Roberto Amadini and Peter J. Stuckey. Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers. In *CP*, 2014.

[Amadini *et al.*, 2014a] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An Enhanced Features Extractor for a Portfolio of Constraint Solvers. In *SAC*, 2014.

[Amadini *et al.*, 2014b] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio Approaches for Constraint Optimization Problems. In *LION*, 2014.

[Amadini *et al.*, 2014c] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: a Lazy Portfolio Approach for Constraint Solving. *TPLP*, 2014.

[Amadini *et al.*, 2015] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP: a Sequential CP Portfolio Solver. In *SAC*, 2015.

[Brucker *et al.*, 1999] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European journal of operational research*, 1999.

[Cenamor *et al.*, 2014] Isabel Cenamor, Tomás de la Rosa, and Fernando Fernández. IBACOP and IBACOP2 Planner, 2014.

[Cire *et al.*, 2014] Andre Cire, Serdar Kadioglu, and Meinolf Sellmann. Parallel restarted search. In *AAAI*, 2014.

[Duda *et al.*, 2000] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.

[Gomes and Selman, 2001] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 2001.

[Hoos *et al.*, 2014] Holger Hoos, Marius Thomas Lindauer, and Torsten Schaub. claspfolio 2: Advances in Algorithm Selection for Answer Set Programming. *TPLP*, 2014.

[Hoos *et al.*, 2015a] Holger H. Hoos, Roland Kaminski, Marius Thomas Lindauer, and Torsten Schaub. aspeed: Solver scheduling via answer set programming. *TPLP*, 2015.

[Hoos *et al.*, 2015b] Holger H. Hoos, Marius Thomas Lindauer, and Frank Hutter. From Sequential Algorithm Selection to Parallel Portfolio Selection. In *LION*, 2015.

[Hurley *et al.*, 2014] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *CPAIOR*, 2014.

[Hutter *et al.*, 2011] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION*, 2011.

[Kadioglu *et al.*, 2010] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - Instance-Specific Algorithm Configuration. In *ECAI*, 2010.

[Kadioglu *et al.*, 2011] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm Selection and Scheduling. In *CP*, 2011.

[Malitsky *et al.*, 2012] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Parallel SAT Solver Selection and Scheduling. In *CP*, 2012.

[Malitsky *et al.*, 2013] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. In *IJCAI*, 2013.

[Nethercote *et al.*, 2007] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP*, 2007.

[O'Mahony *et al.*, 2008] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. *AICS*, 2008.

[Rice, 1976] John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 1976.

[Rossi *et al.*, 2006] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[Roussel, 2011] Olivier Roussel. ppfolio portfolio solver. http://www.cril.univ-artois.fr/~roussel/ppfolio/, 2011.

[Stuckey *et al.*, 2010] Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 2010.

[Valenzano *et al.*, 2012] Richard Anthony Valenzano, Hootan Nakhost, Martin Müller, Jonathan Schaeffer, and Nathan R. Sturtevant. ArvandHerd: Parallel Planning with a Portfolio. In *ECAI*, 2012.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 2008.

[Yun and Epstein, 2012] Xi Yun and Susan L. Epstein. Learning Algorithm Portfolios for Parallel Execution. In *LION*, 2012.