# Parallelizing Constraint Solvers for Hard RCPSP Instances

Roberto Amadini[1], Maurizio Gabbrielli[2], and Jacopo Mauro[3(✉)]

[1] Department of Computing and Information Systems, University of Melbourne, Melbourne, Australia
[2] DISI, University of Bologna, Italy/FOCUS Research Team, INRIA, Rocquencourt, France
[3] Department of Informatics, University of Oslo, Oslo, Norway
mauro.jacopo@gmail.com

**Abstract.** The Resource-Constrained Project Scheduling Problem (RCPSP) is a well-known scheduling problem aimed at minimizing the makespan of a project subject to temporal and resource constraints. In this paper we show that hard RCPSPs can be efficiently tackled by a portfolio approach that combines the strengths of different constraint solvers Our approach seeks to predict and run in parallel the best solvers for a new, unseen RCPSP instance by enabling the bound communication between them. This on-average allows to outperform the oracle solver that always chooses the best available solver for any given instance.

## 1 Introduction

The *Resource-Constrained Project Scheduling Problem* (RCPSP) [10] is the problem of minimizing the makespan (i.e., the total duration) of a project, defined as a collection of tasks subject to precedence relations between the activities and constrained by resource availabilities. This well-known NP-hard problem [9] has countless industrial applications and it is probably one of the most studied scheduling benchmark. The *Constraint Programming* (CP) [18] paradigm allows to model and solve hard combinatorial problems, and in particular the RCPSP can be naturally and elegantly encoded into a *Constraint Optimization Problem* (COP) where: *(i)* integer variables are used to track the start time of each task; *(ii)* constraints over such variables ensure compliance with the precedence relations and resource capabilities; *(iii)* a special integer variable keeps track of the makespan. The goal is to find a consistent assignment of the variables which minimizes the makespan. An effective solving technique to do so is the *Lazy Clause Generation* (LCG) [17] approach. The key idea of LCG is to mimic the Finite Domain propagation by properly generating corresponding SAT clauses during the search.

To exploit the diverse nature and performance of different solving techniques, a fairly recent trend consists in using a portfolio approach [11]. Basically, given

Supported by the EU project FP7-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems.*


P. Festa et al. (Eds.): LION 2016, LNCS 10079, pp. 227–233, 2016.
DOI: 10.1007/978-3-319-50349-3_16

a portfolio $\{s_1, \ldots, s_m\}$ of $m > 1$ solvers, a *portfolio solver* seeks to predict the best solver(s) $s_{i_1}, \ldots, s_{i_k}$ (with $1 \leq i_j \leq m$ for $j = 1, \ldots, k$) for solving a new, unseen problem $p$ and then runs such solver(s) on $p$. Scheduling $k > 1$ solvers can reduce the risk of selecting only one solver and especially enables the knowledge sharing between solvers, as well as their parallel execution. Surprisingly, despite their effectiveness, portfolio solvers have been poorly adopted in real-life applications [5].

In this work we show how CP can be successfully applied for solving hard RCPSP instances by means of a parallel portfolio approach. We retrieved a fairly large number of non-trivial RCPSPs encoded in *MiniZinc* [16] and we defined and test some variants of the parallel portfolio solver `sunny-cp` [3] to boost the resolution of the RCPSP instances. Experimental results show that state-of-the-art LCG solvers can be significantly overcome thanks to other constraint solvers not employing LCG. The message of the paper is twofold: *(i)* we prove that the belief that portfolios can not be applied in real-life scenarios characterized by a dominating solver is false, and *(ii)* we show that by parallelizing the solvers execution and by enabling the bounds communication between the scheduled solvers we can get an overall better solver which is even greater than the sum of its parts. To the best of our knowledge, we are not aware of similar approaches for efficiently solving hard RCPSP instances.

## 2 Background

The RCPSP resolution has attracted a lot of attention over the last decades, since this problem emerges from many real-life scenarios [12–14]. To our knowledge, the LCG approach gives the best results for RCPSP [19] and variants like RCPSP/max [20] and RCPSP/max-cal [15]. In this paper we examine a possible CP formulation of RCPSP, as it appears in MiniZinc 1.6 benchmarks.[1] The CP model is annotated with a default search strategy imposing to select the variable having smaller domain (*min-dom* heuristic) and trying to assign to such variable the smaller value of its domain (*min-value* heuristic). The study of alternative heuristics is outside the scope of this work.

`sunny-cp` is an open-source portfolio solver [3,4]. It enables to run more solvers simultaneously by exploiting their cooperation via bound sharing [6] and restarting policies. `sunny-cp` won the gold medal in the open category of MiniZinc Challenge 2015 and it is currently the only parallel portfolio solver able to solve generic COPs [1]. `sunny-cp` is built on top of SUNNY algorithm [2], which exploits the $k$-Nearest Neighbors algorithm to produce a sequential schedule of solvers for solving a given problem. In a multicore setting, the schedule is parallelized on the $c$ available cores: the first and most promising $c - 1$ solvers are allocated to the first $c - 1$ cores, while the remaining ones are assigned to the last available core. For RCPSPs, the most promising solvers are those that are faster in finding the minimal makespan values in the $k$-neighborhood.

[1] Available at https://github.com/MiniZinc/minizinc-benchmarks/blob/master/rcpsp/rcpsp.mzn.

A *"bound-and-restart"* mechanism is used for enabling the bound sharing between the running solvers. Given a restarting threshold $T_r$, a running solver is stopped and restarted if it has not found a solution in the last $T_r$ seconds and its current best bound is obsolete w.r.t. the overall best bound found by another scheduled solver. `sunny-cp` uses a portfolio of solvers disparate in their nature. Some of them are provided in two variants: *fixed* and *free*. The fixed variant is optional, and forces a solver to use the search strategy possibly defined in the MiniZinc input model. The free variant instead allows a solver to use its preferred search strategy. By default, `sunny-cp` uses the fixed variant. However, as later detailed, the free variant may significantly outperform the fixed one.[2]

## 3 Methodology

The RCPSP model mentioned in Sect. 2 is the most represented problem class in the MiniZinc 1.6 benchmarks with 2904 RCPSP instances coming from different scenarios. However, most of these instances are not challenging: often the best solvers of `sunny-cp` can solve them instantaneously. Therefore, we decided to consider a narrowed dataset $\Delta$ of 647 RCPSPs for each of which no solver can find an optimal solution in 90 s and at least one solver can find a feasible solution.

Fixed a universe of solvers $\mathcal{U}$ and a solving timeout $T$, we measure the performance of solver $s \in \mathcal{U}$ on a problem instance $p \in \Delta$ within $T$ seconds in terms of:

- $\mathsf{OPT}$: measures the optima proven. If $s$ proves the optimality of a solution for $p$, then $\mathsf{OPT}(s,p) = 1$. Otherwise, $\mathsf{OPT}(s,p) = 0$;
- $\mathsf{TIME}$: measures the optimization time. If $s$ proves the optimality of a solution for $p$ in $t < T$ seconds, then $\mathsf{TIME}(s,p) = t$. Otherwise, $\mathsf{TIME}(s,p) = T$;
- $\mathsf{OBJ}$: measures the quality of a solution, by normalizing its makespan value in the range $[0,1]$. If $s$ finds no solution, then $\mathsf{OBJ}(s,p) = 0$. Otherwise, if $\texttt{mkspan}(s,p)$ is the best makespan found by $s$ for problem $p$ and said $\mathcal{M}_p = \{\texttt{mkspan}(s,p) \mid s \in \mathcal{U}\}$, we have: $\mathsf{OBJ}(s,p) = 1 - \dfrac{\texttt{mkspan}(s,p) - \min \mathcal{M}_p}{\max \mathcal{M}_p - \min \mathcal{M}_p}$.

Table 1 shows the average performance of the individual solvers of `sunny-cp` with $T = 900$ s. We added as baseline the Virtual Best Solver (*VBS*), the oracle portfolio solver that —for a given problem and performance metric— always chooses the best solver in the portfolio. As can be seen, for almost 90% of the dataset $\Delta$ (578 instances) no solver is able to prove the optimality in 900 s. Chuffed clearly dominates all the other solvers, almost reaching the *VBS* performance. The effectiveness of LCG is also confirmed by the performance of the others LCG-based solvers, namely CPX and LazyFD. While using the free search is often effective, it is not always the best choice. For this reason we decided to test three different variants of `sunny-cp`:

[2] For more details about `sunny-cp`, we refer the reader to [3].

**Table 1.** Average performance. Fixed version is available only for the solvers marked with *.

| Solver | OPT (%) | | TIME (sec.) | | OBJ × 100 | |
|---|---|---|---|---|---|---|
| | Fixed | Free | Fixed | Free | Fixed | Free |
| Chuffed* | 2.63 | **7.42** | 887.77 | **858.64** | 88.96 | **96.50** |
| G12/CPX* | 1.24 | 1.08 | 894.66 | 894.26 | 89.03 | 75.66 |
| G12/LazyFD* | 0.15 | 2.62 | 899.90 | 888.48 | 72.76 | 75.38 |
| HaifaCSP | - | 0.62 | - | 896.97 | - | 74.73 |
| Choco* | 0 | 0 | 900 | 900 | 66.75 | 72.76 |
| OR-Tools* | 0.31 | 0.16 | 899.05 | 899.14 | 65.00 | 67.11 |
| G12/FD* | 0 | 0 | 900 | 900 | 65.92 | 15.78 |
| Gecode* | 0 | 0 | 900 | 900 | 64.25 | 64.20 |
| MinisatID | - | 0.31 | - | 898.32 | - | 63.64 |
| iZplus* | 0 | 0 | 900 | 900 | 43.43 | 33.74 |
| G12/Gurobi | - | 2.94 | - | 885.90 | - | 2.94 |
| *VBS* | *10.67* | | *841.33* | | *100* | |

- `sunny-def`: the default version of `sunny-cp`. It uses the portfolio $\Pi_{\text{def}}$ of the solvers listed in Table 1 and always chooses the fixed version when available;
- `sunny-all`: uses a portfolio $\Pi_{\text{all}}$ of $11 + 8 = 19$ solvers which extends $\Pi_{\text{def}}$ by including all the versions of all the available solvers;
- `sunny-stc`: uses a variable sized portfolio $\Pi_{c,\mu}$ of $c$ solvers, where $c$ is the number of available cores and $\mu \in \{\mathsf{OPT}, \mathsf{TIME}, \mathsf{OBJ}\}$ is a performance measure. Specifically, $\Pi_{c,\mu}$ is the subset of the best $c$ solvers of $\Pi_{\text{all}}$ according to the average value of $\mu$ over dataset $\Delta$.

Note that both `sunny-all` and `sunny-def` are *dynamic* approaches, since they select the solvers to run on-line according to the instance to be solved. `sunny-stc` follows instead a *static* approach. Indeed, since for each number of cores $c$ its portfolio $\Pi_{c,\mu}$ contains exactly $c$ solvers, no prediction is performed and all its solvers are launched simultaneously regardless of the instance to be solved.

## 4 Results

This Section presents the performance of `sunny-def`, `sunny-all`, and `sunny-stc` in terms of OBJ, OPT, and TIME metrics by considering $1, 2, 4$, and 8 cores. For all the `sunny-cp` variants, we used the default value $T_r = 5\,\text{s}$ for the restarting threshold, and we validated the predictions with a 10-fold cross-validation [7]. In addition to the *VBS* we introduce the *Virtual Parallel Solver* ($VPS_{c,\mu}$), an oracle portfolio solver that for $c \in \{1, 2, 4, 8\}$ and $\mu \in \{\mathsf{OPT}, \mathsf{TIME}, \mathsf{OBJ}\}$ simulates the parallel and independent execution of the

**Table 2.** OBJ performance.

| OBJ × 100 | 1 core | 2 cores | 4 cores | 8 cores |
|---|---|---|---|---|
| `sunny-def` | 92.14 | 94.25 | 95.83 | 96.04 |
| `sunny-all` | 94.67 | 96.36 | **98.25** | 98.45 |
| `sunny-stc` | **94.97** | 95.73 | 97.57 | **99.05** |
| *VPS* | ***94.97*** | ***96.92*** | *97.43* | *98.21* |
| *VBS* | *98.38* | | | |

solvers of the portfolio $\Pi_{c,\mu}$ introduced in Sect. 3. With this definition, the *Single Best Solver* (*SBS*) of the portfolio (i.e., the free version of Chuffed) is equivalent to the $VPS_{1,\mu}$, while $VBS = VPS_{|\Pi_{\text{all}}|,\mu}$. Where there is no ambiguity, we will use the notation *VPS* or $VPS_c$ instead of $VPS_{c,\mu}$.

Table 2 shows OBJ results. We can say that most of the `sunny-cp` approaches provide high quality solutions ($0.95 < \text{OBJ} < 1$) even when optimality is not proven. The only approach that performs rather poorly is `sunny-def` with one core. For $c \geq 4$ the effectiveness of bounds communication becomes clear. `sunny-stc` with 4 cores is better than $VPS_4$, i.e., its corresponding version without bounds communication and synchronization issues. With 8 cores `sunny-stc` outperforms not only $VPS_8$, but also the *VBS*. In other terms, 8 cores are enough for providing better solution than a "magic solver" that runs simultaneously —without synchronization issues— all the 19 solvers of $\Pi_{\text{all}}$. The peak performance is reached by `sunny-stc` with 8 cores: the *VBS* is outperformed 158 times (24.42% of $\Delta$), meaning that almost one time out of four it finds a better solution than *VBS*.

The OPT performance is depicted in Table 3. This metric is challenging in our context: we are dealing with hard RCPSP instances for which no solver is able to prove the optimality in less than 90 s and the *VBS* can prove only 69 optimum (10.66% of $\Delta$). For $c \geq 2$ the best approach is `sunny-all`, which is able to outperform the *VBS* whith 4 or more cores. In particular, the gain with 8 cores is somewhat impressive: 4.8% optima proven more than *VBS*. Here the performance difference is not only due to parallelism and bounds communication, but especially due to the solver selection. Indeed, the gap with `sunny-stc` becomes

**Table 3.** OPT performance.

| OPT (%) | 1 core | 2 cores | 4 cores | 8 cores |
|---|---|---|---|---|
| `sunny-def` | 2.01 | 4.64 | 9.58 | 10.36 |
| `sunny-all` | 6.65 | **9.58** | **12.52** | **15.46** |
| `sunny-stc` | **7.42** | 7.88 | 9.12 | 9.58 |
| *VPS* | ***7.42*** | *8.04* | *8.35* | *8.50* |
| *VBS* | *10.66* | | | |

**Table 4.** TIME performance.

| TIME (sec.) | 1 core | 2 cores | 4 cores | 8 cores |
|---|---|---|---|---|
| `sunny-def` | 889.33 | 869.42 | 830.71 | 825.55 |
| `sunny-all` | **858.20** | **838.41** | **823.18** | **797.53** |
| `sunny-stc` | 858.63 | 853.55 | 849.45 | 844.69 |
| *VPS* | *858.63* | *854.13* | *852.16* | *851.17* |
| *VBS* | *841.33* | | | |

larger as the number of cores increases. Table 4 shows instead the average TIME performances. Being the majority of the instances of $\Delta$ very hard to solve, it is not surprising that the TIME values are very close to the timeout $T = 900$. All the tested approaches perform well, since they are very close to, or better than, the *VBS*. The effectiveness in reducing the optimization time is also corroborated by the fact that for 41 instances (6.33% of $\Delta$) `sunny-all` can prove the optimality of a solution in less than 90 s.

Summarizing, we can say that all the `sunny-cp` variants we tested can be effective on the RCPSP instances of $\Delta$, especially when more than one core is used. The solver's parallelization is not the only key for the success of such approaches. The use of the free search and the bounds communication between the scheduled solvers enable to outperform the *VBS*. Furthermore, it is also important —especially for OPT and TIME metrics— to properly schedule a subset of solvers dynamically, i.e., according to the instance to be solved. For a more in depth discussion of the results and more data we invite the interested reader to the companion technical report available at https://hal.inria.fr/hal-01295061.

## 5 Conclusions

The Resource-Constrained Project Scheduling Problem (RCPSP) is a well-known scheduling problem applicable in many real-life scenarios. In this paper we show how it is possible to boost its resolution by using a portfolio of different constraint solvers for selecting and running a subset of them on multiple cores. Improvements are manifold in terms of both solution quality, optima proven and optimization time. We noticed in particular significant performance gains in quickly proving more optima.

We believe that this work may open the way to several extensions. An interesting one concerns the analysis of how to properly integrate and combine the concurrent execution of different constraint solvers. It is also certainly worth to evaluate and combine other search heuristics for the RCPSP resolution (e.g., precedence-setting searches, texture-based heuristics [8]). From the portfolio perspective, we hope that this work can stimulate the utilization of portfolio solvers also in real-life scenarios where typically a single, dominant solver is used for solving different instances of the same problem.

## References


1. Amadini, R., Gabbrielli, M., Mauro, J.: Portfolio approaches for constraint optimization problems. In: Pardalos, P.M., Resende, M.G.C., Vogiatzis, C., Walteros, J.L. (eds.) LION 2014. LNCS, vol. 8426, pp. 21–35. Springer, Heidelberg (2014). doi:10.1007/978-3-319-09584-4_3
2. Amadini, R., Gabbrielli, M., Mauro, J.: SUNNY: a Lazy portfolio approach for constraint solving. TPLP **4–5**, 509–524 (2014)
3. Amadini, R., Gabbrielli, M., Mauro, J.: A multicore tool for constraint solving. In: IJCAI, pp. 232–238 (2015)
4. Amadini, R., Gabbrielli, M., Mauro, J.: SUNNY-CP: a sequential CP portfolio solver. In: SAC, pp. 1861–1867 (2015)
5. Amadini, R., Gabbrielli, M., Mauro, J.: Why CP portfolio solvers are (under)utilized? Issues and challenges. In: Falaschi, M. (ed.) LOPSTR 2015. LNCS, vol. 9527, pp. 349–364. Springer, Heidelberg (2015). doi:10.1007/978-3-319-27436-2_21
6. Amadini, R., Stuckey, P.J.: Sequential time splitting and bounds communication for a portfolio of optimization solvers. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 108–124. Springer, Heidelberg (2014). doi:10.1007/978-3-319-10428-7_11
7. Arlot, S., Celisse, A.: A survey of cross-validation procedures for model selection. Statist. Surv. **4**, 40–79 (2010)
8. Christopher Beck, J., Davenport, A.J., Sitarski, E.M., Fox, M.S.: Texture-based heuristics for scheduling revisited. In: AAAI, pp. 241–248 (1997)
9. Blazewicz, J., Lenstra, J.K., Rinnooy Kan, A.H.G.: Scheduling subject to resource constraints: classification and complexity. Discret. Appl. Math. **5**(1), 11–24 (1983)
10. Brucker, P., Drexl, A., Mohring, R.H., Neumann, K., Pesch, E.: Resource-constrained project scheduling: notation, classification, models, and methods. Eur. J. Oper. Res. **1**, 3–41 (1999)
11. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **1–2**, 43–62 (2001)
12. Hartmann, S., Briskorn, D.: A survey of variants and extensions of the resource-constrained project scheduling problem. Eur. J. Oper. Res. **1**, 1–14 (2010)
13. Herroelen, W., De Reyck, B., Demeulemeester, E.: Resource-constrained project scheduling: a survey of recent developments. Comput. OR **4**, 279–302 (1998)
14. Kolisch, R., Hartmann, S.: Experimental investigation of heuristics for resource-constrained project scheduling: an update. Eur. J. Oper. Res. **1**, 23–37 (2006)
15. Kreter, S., Schutt, A., Stuckey, P.J.: Modeling and solving project scheduling with calendars. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 262–278. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23219-5_19
16. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: CP, pp. 529–543 (2007)
17. Ohrimenko, O., Stuckey, P.J., Michael, C.: Propagation via lazy clause generation. Constraints **3**, 357–391 (2009)
18. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming (2006)
19. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. Constraints **3**, 250–282 (2011)
20. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Solving RCPSP/max by lazy clause generation. J. Sched. **3**, 273–289 (2013)