Parallelizing Constraint Programming with Learning

Thorsten Ehlers $^{\star 1}$ and Peter J. Stuckey $^{\star\star 2,3}$

[1] Kiel University, Department of Computer Science
D-24098 Kiel, Germany,
`the@informatik.uni-kiel.de`

[2] Department of Computing and Information Systems
University of Melbourne, 3010 Australia
`pstuckey@unimelb.edu.au`

[3] National ICT Australia, Victoria Laboratory

Abstract. Parallel Constraint Programming (CP) solvers typically split the search space in disjoint subspaces, and run solvers independently on these. This may induce significant overhead when solving optimization problems. Parallel Boolean Satisfiability (SAT) solvers typically run a portfolio of solvers, all solving the same problem but sharing some limited learnt clause information. In this paper we consider parallelizing a lazy clause generation (LCG) constraint programming solver, which is a constraint programming solver with learning. Since it is both a kind of CP solver and a kind of SAT solver it is not clear which approach to parallelization is likely to be most effective. We give examples of very different kinds of optimization problems we wish to parallelize and show that a hybrid approach to parallelization can provide a robust and high performing parallel LCG solver.

1 Introduction

Techniques for verification and optimization such as SAT, CP, SMT and MIP have greatly improved in the last decades, and are nowadays used in a wide range of applications. Besides algorithmic improvements, more and more powerful hardware has become available, giving an additional boost on sequential performance. But the time of this free lunch seems to be over, as clock rates and instructions per cycle are hardly improving anymore. In order to gain speedups from today's hardware, algorithms should be able to run in parallel. In this paper, we consider the parallelization of the LCG solver Chuffed [5] for CP-based optimization problems. Chuffed combines CP techniques such as search and

* Supported by a fellowship within the FITweltweit program of the German Academic Exchange Service (DAAD).

** NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

strong propagation with techniques developed for SAT solving such as clause learning, restarts and activity based search.

Whereas parallelization of CP solvers is usually based on some kind of search space splitting, parallelization of SAT solvers is usually based on some form of portfolio approach. Hence an interesting question arises for LCG solvers: should they use search space splitting or portfolio methods for parallelizing search? In this paper we investigate this question.

The contributions of this paper are

1. An analysis of the runtime-behavior of sequential solvers on optimization problems, showing extremely different characteristics of different problems.
2. An optimistic branching technique which allows for finding good solutions much earlier, which prevents superfluous work in search space splitting.
3. A comparison of search space splitting and work stealing, the common approach used in parallel CP, with a portfolio CP solver using techniques commonly used in parallel SAT.
4. A scalable, parallel LCG solver which allows for significant speedups on a wide range of benchmarks. Compared to the sequential solver, superlinear speedup is achieved in finding good solutions.

The structure of the paper is as follows. After discussing related work in Section 2 and presenting the architecture of our parallel solver (3), we then examine the use of sequential optimization on two very different optimization problems in Section 4 and show the impact of basic approaches to parallelizing their solving. In Sections 5 and 6, we present results on a suite of benchmarks for parallelizing using search space splitting implemented by work stealing, and an approach based on SAT-like portfolio solving. We then consider the effect of splitting the problem by objective value in Section 7. After combining these approaches to a stable and scalable solver in Section 8, we conclude in Section 9.

2 Related Work

The most common approach for solving CSP problems is to combine search with propagation [27]. The search is implemented as backtracking, and at each node of the search tree propagators are invoked to reduce variable domains with respect to the decisions made during branching. In case an inconsistent state is detected, i.e. some variable can take no possible value, the solver backtracks, and tries another variable assignment. Implementing fast and scalable parallel algorithms is noted as one of the large challenges in optimization [9].

2.1 Parallel CP

Parallel algorithms for CP typically split the search space, and run solver threads on disjoint subspaces [11]. This approach has been studied for several decades, and it is known that superlinear speedups are possible in some cases [14,22]. Most solvers use work stealing mechanisms to keep all solver threads busy [26], and significant speedups are reported for up to 512 threads [15]. Recent research tried to reduce the communication overhead in order to improve speedups for massively-parallel search. In [17], the authors suggest to split the search space by computing the discrepancy from a given search strategy. Thus, solver threads only need to know about their index to compute their chunk of the search space. Unfortunately, they report results for only a small number of experiments, and cannot prove optimality for them. Another approach is to split the search space before starting the search, and storing chunks of work in a master process [23,24].

If a good search strategy for a specific problem is known, this may be used to focus the parallel search on promising parts of the search space [6], and gain significant speedups.

It is known that some solvers are faster than others, depending on the problem instance. This fact was used in [2] to build a sequential portfolio, and later to create a parallel portfolio solver [1]. Significant speedups could be gained using a small number of threads, but it is not clear how scalable this approach is.

2.2 Parallel SAT

The satisfiability problem of propositional logic (SAT) can be seen as a special case of CP with binary variables, and constraints given in form of disjunctions, called clauses. It is typically solved using conflict driven clause learning (CDCL), an extension of the well-known DPLL algorithms, together with agile restarting strategies and activity based search [18]. These techniques allow for reusing information about parts of the search space which were proven infeasible, and restarting the search to emphasize important variables as well as recovering from bad decisions made close to the root of the search tree. Parallel algorithms for SAT either split the search space, or run different solver configurations in parallel [3]. The latter approach, typically referred to as portfolio approach, has proven very successful especially on structured instances. Recent research focusses on the exchange of learnt clauses between solver threads [7,12]. Unfortunately, the scalability of these solvers seems to be limited due to the sequential structure of resolution proofs [13].

2.3 Lazy Clause Generation

Lazy Clause Generation (LCG) [8] combines techniques from CP and SAT to solve Constraint Satisfaction Problems. If an inconsistency is detected during search, the reasons for this inconsistency are compiled into a clause, and added to the set of constraints. Thus, it is possible to reuse this knowledge in other parts of the search tree, which is extremely helpful if these clauses are good explanations for the failed search [28]. The LCG-based solver CHUFFED[4] [5] additionally supports the Variable State Independent Decaying Sum (VSIDS) branching heuristic that is commonly used in SAT solvers. This heuristic branches on variables first that have recently been involved in conflicts. CHUFFED can switch between activity based search and programmed search during runtime.

[4] https://github.com/geoffchu/chuffed

3 Preliminaries

A Constraint Satisfaction Problem (CSP) problem ϕ is given by a triple (V, D, C), where $V = (v_1, \ldots, v_n) = vars(\phi)$ is a set of n variables on finite domains $D = (D_1, \ldots, D_n)$, and C is a set of predicates $c : D \mapsto \{\bot, \top\}$. We will assume integer variables, i.e. $v \in [lb(v), ub(v)] \cap \mathbb{Z}$ for all variables v. The set of feasible solutions is given by $S = \{x \in D \mid \forall c \in C.c(x)\}$. A Constraint Optimization Problem (COP) is a tuple (V, D, C, z) consisting of a CSP (V, D, C) and a variable z that takes the objective value. Throughout this paper we will only consider minimization problems, as maximization can be expressed by negating the expression that z is equated to. When adding further constraints, we will write $\phi_{|c} = (V, D, C \cup \{c\})$.

3.1 Parallel Solver Architecture

We have developed a parallel version of Chuffed [5] which is used in all experiments. Chuffed is a state-of-the-art lazy clause generation solver [20]. It comes with a Master-Slave-infrastructure [6], where communication is performed as message passing via MPI. When gaining parallelism by search space splitting, the master process sends conjunctions of literals, called jobs, to the slaves. We extended this scheme as follows to gain a more flexible solver.

– **Portfolio-Solving**, as in parallel SAT solving, can be achieved by sending empty jobs to each slave process, which allows them to search the whole search space. Diversification is gained by initializing the VSIDS-activities with random values.
– **Probing** on variable values: The master process can send jobs of the form $[x \circ c]$ to the slave processes, where $x \in vars(\phi)$, $c \in \mathbb{Z}$, and $\circ \in \{=, \neq, \leq, <, >, \geq\}$. This can, e.g., be used for guessing bounds on the objective value.
– **Learnt clauses** are sent to, and forwarded by the master process, if their length is sufficiently small. The threshold on the length can be adjusted dynamically to both maintain a sufficient communication between solvers, and prevent network congestion.
– **Adaptive** size of clause database: While Chuffed has a fixed bound on the size of its clause database, we allow for a dynamic amount of received clauses. Whenever the learnt clause database is cleaned, we delete all received clauses with low activity.
– **Hybrid** approaches: It is possible to mix the modi operandi.

4 Optimization

As CP typically deals with decision problems, CP-based optimization is built around decision procedures. Figure 1 shows how, given a decision procedure Decide, an optimization algorithm can use this procedure to find an optimum solution. Running this algorithm will result in a sequence of solver calls, of which

Algorithm 1 Optimize CP

```
function OPTIMIZE($\phi, z, lb, ub$)
    $res \leftarrow (\bot, \infty)$
    while ($ub \geq lb$)
        $(sat, x) \leftarrow Decide(\phi_{|z \leq ub})$
        if ($sat$) $res \leftarrow (\top, z)$, $ub \leftarrow z - 1$
        else break
    return $res$
```

the last one returns UNSAT, proving that either no solution exists, or that the last solution found is optimal. In the remainder of this paper, we refer to the last call as the "proving optimality" part, and all other calls as the "search" part of the overall run.

In this section, we examine the behavior of these algorithms on two examples, both for the sequential and parallel case. The examples, `cargo`[5] and `mqueens`[6] were chosen from the MiniZinc Challenges 2013 and 2014, respectively. In all these experiments, the free search of CHUFFED was used: On every restart, CHUFFED flips between programmed search and VSIDS (restart_flip from [25]).

4.1 Parallel Optimization

We begin with the naïve approach, and simply reuse a parallel solver for optimization problems. Whenever one of the parallel solvers finds a solution, this is reported to the master. Furthermore, if the objective value of this solution is c, a unit clause containing the literal $[z < c]$ is sent to all solvers for stronger pruning. We parallelized solving using a portfolio approach, and using search space splitting with work stealing [6]. For the portfolio approach, the search is diversified by initializing variable activities with random values, based on different seeds for each solver thread, as is common in SAT portfolios.

Figure 1 shows the development of the objective value during the solver run on the `cargo` benchmark for $p \in \{2, 8, 64\}$ processes. Note that we re-use the master-slave architecture for this experiment, thus, one of these processes denotes the master. The speed-up we observe is very limited: 2.2 for 7 worker processes, and 4.2 for 63 workers. The reason for this is simple: All of the parallel processes find many solutions independently of each other, but they hardly benefit from new bounds found by other solvers. Similar results can be observed when gaining parallelism by splitting the search space, c.f. Figure 2. Here, a lot of solutions are found in disjoint parts of the search space. Exchanging bounds on the objective among the worker threads leads to small improvements of the running time, but the overall speedup is disappointing.

Our second running example, `mqueens`, shows a different behavior. Here, the running time is dominated by proving optimality, which is proving unsatisfiabil-

[5] http://www.minizinc.org/challenge2013/probs/cargo/challenge04_1s_626.dzn
[6] http://www.minizinc.org/challenge2014/probs/mqueens/n12.dzn

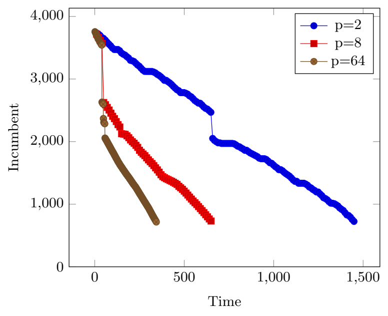

Fig. 1. `cargo`: results for portfolio parallel solving

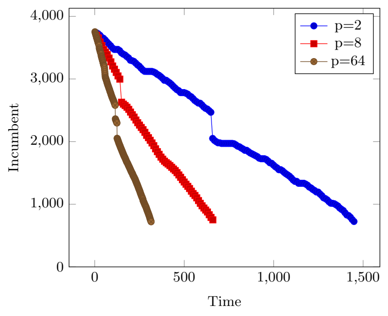

Fig. 2. `cargo`: results for search space splitting parallel solving

# CPUs	Workers	`Portfolio`	`SSS`
2	1	155s	155s
8	7	102s	22s
64	63	39s	11s

Fig. 3. `mqueens`: run times.

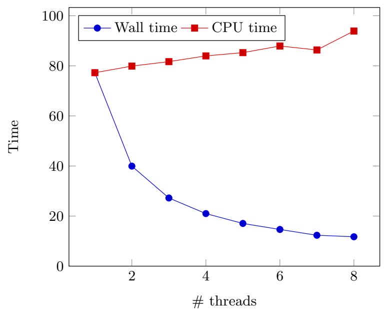

Fig. 4. `mqueens`: scaling behavior of Gecode

ity. The portfolio solver shows some, but very limited speedup. Apparently, there is no small optimality proof here, and exchanging clauses among the solvers is of limited success due to the sequential structure of resolution proofs [13]. Search space splitting is much more promising here, with a speedup of 6.9 when using 7 workers threads instead of 1. Unfortunately, using more cores yields only limited additional speedup. Almost linear speedups can also be observed when running the parallel version of the CP solver Gecode on this benchmark, c.f. Figure 4. Gecode also uses search space splitting to gain parallelism, and work stealing for load balancing.

We summarize these experiments with two main observations. First, search space splitting is superior to portfolio solving in terms of proving optimality. When run on many cores, it is crucial to split the work space in equally hard chunks to benefit from more parallel threads. Second, both approaches scale poorly when many suboptimal solutions can be found.

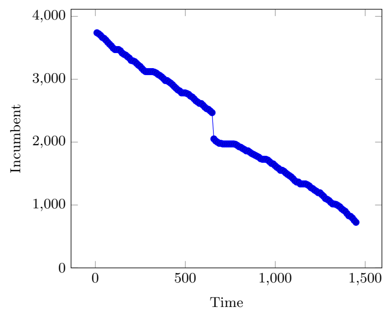

Fig. 5. `cargo`: slow convergence towards optimal result.

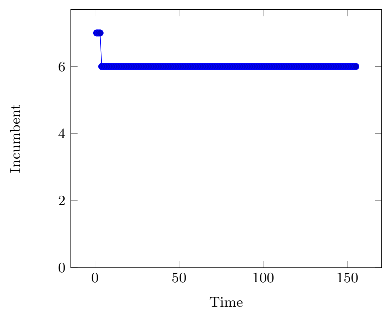

Fig. 6. `mqueens`: slow proof of optimality.

4.2 Sequential Optimization

To gain a better understanding of the results we found in the parallel setting, we discuss results obtained by running the sequential solver. For the two benchmarks `cargo` and `mqueens`, we run the sequential version of Chuffed, and periodically record the best objective value found so far. The results for `cargo` can be seen in Figure 5: starting with a value of $3,757$, the objective value is improved steadily, and drops to the optimum of 714 after $1,453$ seconds. In this case, the solver spends the vast majority of the running time improving the solution, and finds $2,654$ different solutions before proving optimality. Interestingly, the improvement speed is roughly constant during the whole run, and the final call to the decision procedure, which proves optimality, is not harder than previous calls. For `mqueens`, we observe a totally different behavior, c.f. Figure 6. An optimum solution is found within 4 seconds, whereas the proof of optimality takes another 151 seconds.

This behavior is reflected by the difficulty of the respective decision problems. For both benchmarks, we added different bounds on the objective, and aborted the solver after finding the first solution. For `cargo` (Figure 7), we ran this experiments for objective values in the interval $[700, 1000]$, i.e. for values close to the optimum objective value. The maximum running time observed was 30s, and average running times of 3 seconds. A very interesting result of this experiment is that the running time close to the optimum solution is not higher than running times for higher bounds. So if we knew a good bound on the objective in advance, we might run the solver with a tightened bound, and find an optimal solution much faster. The `mqueens` benchmark shows a totally different behavior, c.f. Figure 8. Here, the proof of optimality is hard, whereas both finding solutions and proving bounds tighter than the optimal value is extremely fast.

5 Search Space Splitting

As shown in Section 4, parallelisation by splitting the search space in disjoint parts allows for very good speedups, especially for proving optimality. Unfortu-

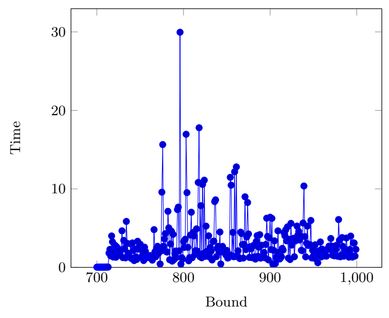

Fig. 7. `cargo`: time for decision problem, depending on bound on objective value.

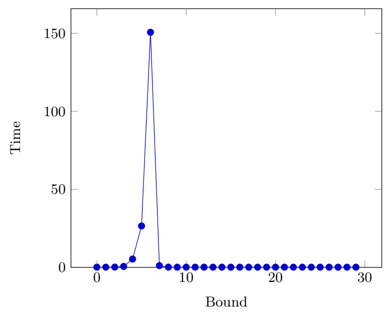

Fig. 8. `mqueens`: time for decision problem, depending on bound on objective value.

nately, this approach comes with some drawbacks. After finding a solution, a sequential algorithm will continue its search, using a tighter bound on the objective value for further pruning. In parallel, worker threads may therefore search parts of the search space that would not be searched by a sequential algorithm, which may dramatically decrease the efficiency, as mentioned in [6].

In order to keep waiting times of worker threads low, it is common to store some jobs, i.e. chunks of the search space, at the master process. Whenever a worker finishes working on its part of the search space, a new chunk of work can be provided without waiting for another worker to provide work. In the worst case, this can lead to situations in which none of the workers searches the part of the search space containing the optimum solution. The decision how to split the search space is very important for gaining some benefit from this approach: Let $\phi = (V, D, C)$ denote an unsatisfiable CSP, and $C' \subset C$ a minimum unsatisfiable core, i.e. a set of constraints such that (V, D, C') is already unsatisfiable. Splitting on a variable that does not occur in C' will then be less likely to speed up the parallel solver. To overcome this problem, VSIDS activities can be used to choose variables for splitting the search space. As VSIDS focusses on variables that were involved in conflicts, this prevents branching on uninteresting variables. In our implementation, the master sends the empty job to one worker, which starts to work on this job. Whenever work has to be stolen, the master sends a request to one of the slaves, which creates new jobs according to its topmost branching decisions.

Example 1 *Assume a worker is working on a job given as $x_1 \wedge x_2$, and its topmost branching decisions are x_3 and x_4. If asked to provide two new jobs, it fixes its new job to $x_1 \wedge x_2 \wedge x_3 \wedge x_4$, and reports this to the master. In turn, the master creates new jobs $x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4$ and $x_1 \wedge x_2 \wedge \bar{x}_3$.*

LCG solvers can reuse information about failed search to further prune the search space. Thus, the time required by a LCG solver to refute a part of the search space depends on previous search. If parallelism is gained by splitting the search space and running LCG solvers on disjoint parts of it, this may decrease

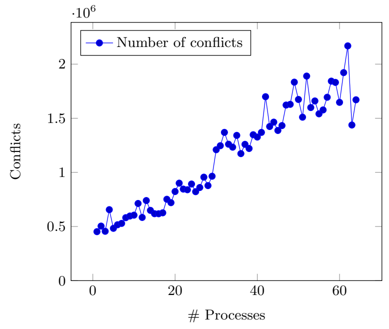

Fig. 9. `mqueens`: number of conflicts in search space splitting parallel solving without clause sharing

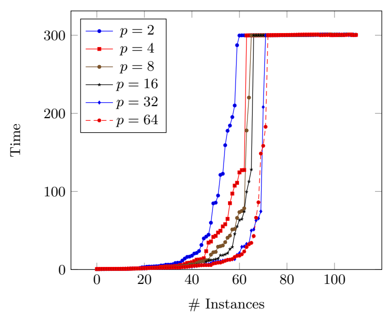

Fig. 10. `suite`: scaling of `SSS` with number of cores

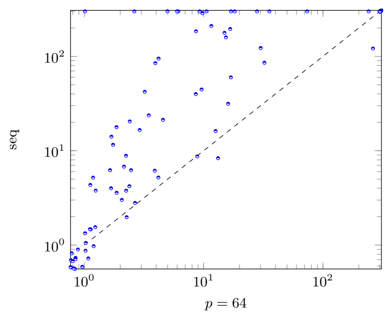

Fig. 11. `suite`: comparison between sequential and `SSS` with 64 cores

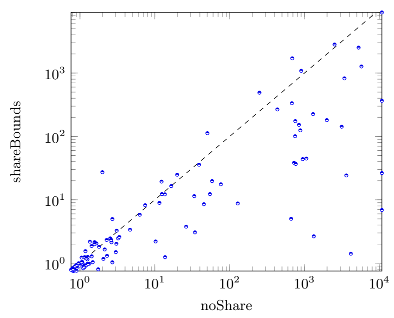

Fig. 12. `suite`: impact of bounds sharing in `SSS` with 64 cores

the achievable speedup: whenever a worker receives a new chunk of the work space, it needs to learn clauses which are relevant to this new subspace, which might be the same as clauses for other chunks of the work space. Figure 9 shows the total number of conflicts occurring while solving `mqueens`. Here, the number of conflicts increases with additional processes. To reduce this burden, we exchange learnt clauses between solvers.

Unless something different is stated for single experiments, this solver exchanges bounds on the objective, and short clauses. Short learnt clauses are exchanged between the solver processes. The bound on their size is adjusted dynamically such that approximately 10% of the clauses in the database are received from other solvers, which gave good results in our tests. As in [12], we check the number of imported clauses regularly, and adjust the threshold on clause size to exchange if too many or too few clauses were received.

To evaluate parallelization approaches we created a set of 110 different benchmarks, `suite`, taken from MiniZinc challenges for 2013 and 2014.[7] We used a

[7] The exact set of instances is available at people.unimelb.edu.au/pstuckey/pchuffed.

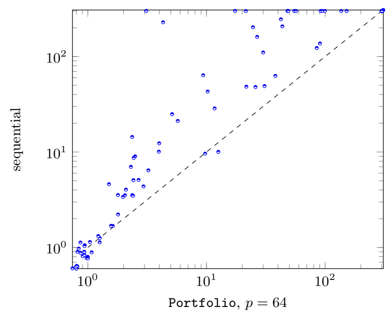

Fig. 13. `suite`: comparison between sequential and `Portfolio`

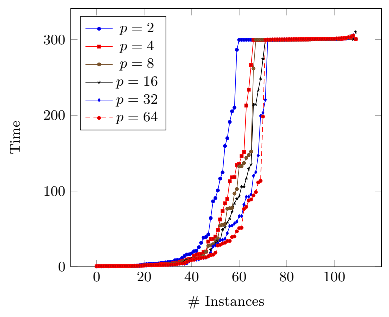

Fig. 14. `suite`: scaling of `Portfolio` with number of cores

time limit of 5 minutes. We denote by `SSS` our search space splitting parallel solver. Figure 10 shows the scaling behavior for 2 to 64 processes. Significant speedups can be observed for up to 32 parallel processes. In Figure 11, we compare the results for a parallel solver on 64 cores with the ones obtained using the sequential solver. The parallel solver clearly outperforms the sequential version. Furthermore, 12 more instances can be solved to optimality within 5 minutes.

In Figure 12, we compare the running times with and without exchanging bounds on the objective between the solver processes. In some cases, e.g. if a good solutions can be found by all of the parallel solvers, there is only a small difference, whereas there is a huge difference for other instances, and 3 more benchmarks can be solved to optimality. In other words, it is crucial for the performance of a parallel LCG solver to find and communicate good bounds on the objective value as fast as possible.

Note that the benchmarks which timed out do not mean an equal result: As we are dealing with optimization problems, they often time out with different incumbent solutions. This issue will be further considered in Section 7.

6 Portfolio

Portfolio solving is a common approach for parallel SAT solving. In this section, we investigate the behavior of a portfolio CP-solver with learning. As common in parallel SAT, the solvers are diversified by initializing their variable activities randomly. Additionally, we allow for some communication between the solver processes. As we are using a master-slave-architecture, a portfolio approach is simulated by sending the empty job (i.e. the empty conjunction) to each solver. For clause exchange, the same policy as in the `SSS` setting is used. We denote this solver as `Portfolio`. Figure 13 compares the running times of the sequential solver, and the portfolio solver on 64 cores, with a time limit of 5 minutes. Only little speedup can be observed for easy instances, whereas parallelism pays off for harder instances, and results in 10 more solved instances. The scaling behavior is shown in Figure 14. Both significant speedups and an increased number of

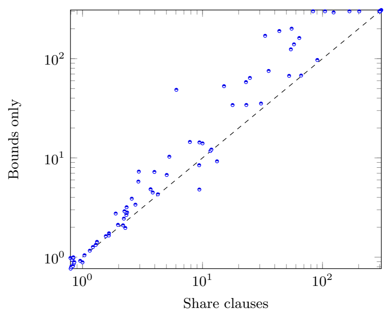

Fig. 15. `suite`: impact of clause sharing on `Portfolio` with 64 cores

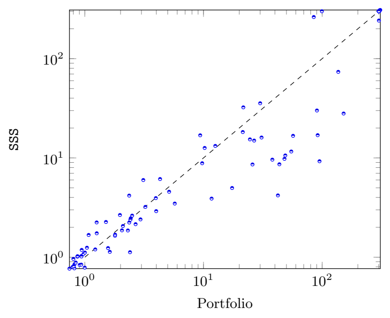

Fig. 16. `suite`: comparison of `SSS` and `Portfolio`, 64 cores

solved instances can be observed when using more CPU cores. On the other hand, 36 of the benchmarks time out, so either no optimum result was found, or the proof of optimality could not be completed.

In parallel SAT solving, it is a well-known fact that clause exchange is very helpful, especially for unsatisfiable instances. For parallel LCG, communication is also beneficial, but it appears harder to determine which, and how many clauses should be exchanged. Figure 15 compares the results of a portfolio solver on 64 cores with our adaptive clause exchange policy to a portfolio solver which only exchanges the incumbent objective value. Communicating learnt clauses yields a significant speedup and 6 more solved instances. Although this is a significant improvement, the power of clause exchange for parallel LCG appears limited. Further experiments showed that the exchange of clauses of size at most 2 speeds up the computation, whereas larger clauses do not always help, and may even significantly impede solving. As it appears difficult to determine the right choice of clauses to exchange, we used the conservative, adaptive approach. Recent work in SAT has emphasized the fact that many learnt clauses are not helpful for satisfiable formulas [19]. As the optimization process consists of solving a sequence of satisfiable problems, followed by one unsatisfiable one—the proof of optimality—this may be the reason for these results. Nevertheless, the portfolio solver is surprisingly strong. Using 64 cores, it can solve one instance that cannot be solved optimally by the search space splitting solver, c.f. Figure 16.

7 Objective Probing

Both search space splitting and portfolio approaches yield good results. Nevertheless, they do not make use of the following observations from Section 4: in many cases finding a good solution is not harder than finding any solution. Finding good solutions early prunes the search, using the tighter objective bounds, and conversely, finding them late result in superfluous search, and may reduce the benefits of parallelism, c.f. Figure 12. Hence, it appears promising to try to push a parallel solver towards finding good solutions quickly.

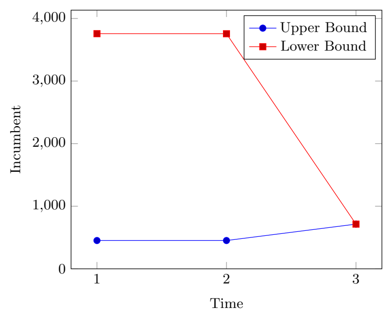

Fig. 17. `cargo`: objective probing portfolio solver

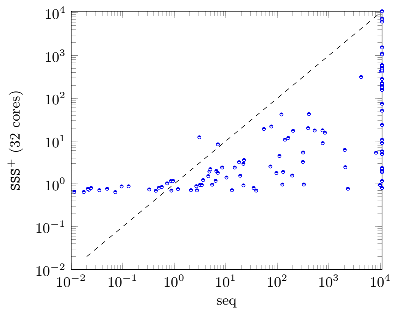

Fig. 18. `suite`: speedup in finding good solutions with objective probing $\mathtt{SSS}^+$

To do so, we guess bounds on the objective, and use some solver processes to probe whether there exists a solution satisfying the guessed bound, or not. A similar approach was already used for parallel Boolean optimization in [16], where the authors compute upper and lower bounds concurrently. When using n processes for objective probing, we use bounds

$$bound(i) = lb + \left\lfloor \frac{i(ub - lb)}{n} \right\rfloor, \tag{1}$$

where lb and ub denote lower and upper bounds on the objective value, and the i^{th} process solves $\textsc{Optimize}(\phi, z, lb, bound(i))$.

Figure 17 shows the impact of this approach when solving the `cargo` benchmark. Compared to the naïve portfolio approach, c.f. Figure 1, an impressive speedup of 483 is achieved, as the solver finds an optimum solution and proves its optimality within 3 seconds. Furthermore, probing objective values yields lower bounds on the objective. Thus, this approach allows for estimating the quality of solutions.

In the remainder of this section, we will discuss how to implement the objective probing, and show results. As we deal with optimization problems, we also consider the quality of solutions found. Therefore, we ran the sequential version of Chuffed on each benchmark with a time limit of 3 hours, and recorded the best solution found. Then, we tested how long it takes the parallel solver to find a better solution, or prove that no better solution exists.

7.1 Objective Probing in Search Space Splitting

For the search space splitting solver, we split the workers in three groups of equal size. Workers from the first group run on split parts of the search space as before. Workers from the second subset start by guessing an objective value according to equation 1. If this guess is refuted, i.e. a proof is found that no solution exists with an objective value satisfying the bound, or it is implied, i.e. a better solution is found, they join the workers from the first group. The

Table 1. `suite`: speedups when searching for good solutions.

#CPUs	SSS				SSS$^+$			
	all		hard		all		hard	
	avg	median	avg	median	avg	median	avg	median
4	2.8	2	10.6	4.5	3.7	3.2	15.5	7.4
8	5	3.8	25.5	9.7	6.2	4	41.8	20.2
16	6.7	5.9	41.2	19.5	9.6	7.6	78.9	34.6
32	9.6	8	72.5	58.9	12.7	13.3	121.3	58.5
64	12.7	15	136.8	104	15.6	13.8	193.8	107

remaining threads behave like the ones from the second group, but re-guess bounds on the objective, until half of the given time limit is reached. This can be seen as a hyper-binary search on the objective, as the interval between lower and upper bound is split in several parts. We denote this solver as SSS$^+$.

Figure 18 shows the impact of this technique on the search for good solutions. For very easy instances, which can be solved in less than one second, the parallel solver is slower than the sequential version. For harder instances, significant speedups can be observed. Table 1 shows the geometric average, and median speedups obtained on all benchmarks, and on hard ones. Here, a benchmark is considered hard if the sequential Chuffed does not terminate within 300 seconds. The average speedup on all instances is sublinear, as many of them are too easy and do not allow for sufficient speedups by the parallel solver. Conversely, the speedup on hard instances is significant, and superlinear for every configuration. The configuration which uses objective probing, SSS$^+$, reaches an average speedup of 193.8 on 64 cores. On 8 and 16 cores, it is even faster than SSS on 16 and 32 cores, respectively. On benchmarks of medium difficulty, the results are mixed. Figure 21 compares the results of the SSS and SSS$^+$ configuration, when using 64 cores. The SSS$^+$ configuration is significantly faster on some benchmarks, and slightly slower on some others. This is especially the case if probing fails in many cases, and thus only yields improved lower bounds on the objective instead of tighter pruning. Summarizing, combining a Search Space Splitting solver with objective probing gives an additional boost on the performance, especially for hard problems with a large value range for the objective value. Splitting the solvers in three groups of equal size works well on our benchmark suite, and it appears hard to find better choices that work well for all benchmarks, or adapt the group size dynamically.

7.2 Portfolio Solving and Objective Probing

In portfolio solving, guessing bounds on the objective value may be seen as an additional source of diversification for the solvers. As in the SSS$^+$ configuration, we split the solver processes in three groups. After objective probing is finished, the respective solvers continue running as in the normal portfolio configuration.

On benchmarks of medium difficulty, this approach outperforms the common portfolio configurations, as can be seen in Figure 20.

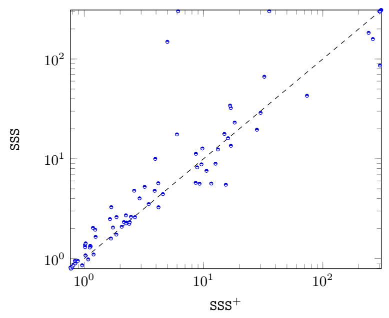

Fig. 19. `suite`: comparison between `SSS` and `SSS`$^+$

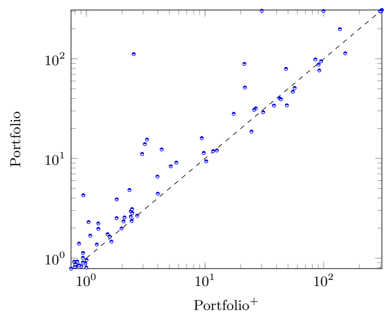

Fig. 20. `suite`: comparison between `Portfolio` and `Portfolio`$^+$

Table 2. `suite`: speedups when searching for good solutions.

#CPUs	`Portfolio`				`Portfolio`$^+$			
	all		hard		all		hard	
	avg	median	avg	median	avg	median	avg	median
4	3.2	2	13.4	4.3	4.3	2.4	18.2	7.4
8	4.7	3.7	23.5	9	6.1	4.2	33.9	8.43
16	6.1	4.5	39.4	14.6	9.2	7	68.4	26.9
32	7.7	7.4	62.1	38	11.4	8.9	107.5	77.4
64	9.1	10.1	84.6	42	13.6	14.6	152	133.6

The reason for this behavior seems to be the following: The portfolio approach is fast in finding good solutions, but for proving optimality it does not scale as well as the search space splitting solver. Thus, the solving process is accelerated if better solutions are found early, but it is not slowed down too much if some workers spend computation time on proving lower bounds instead of participating in the proof of optimality.

Table 2 shows the speedups obtained when searching for good solutions. Again, superlinear average speedups can be observed for all configurations when considering only the hard benchmarks, reaching a maximum of 152 for the portfolio solver with objective probing, denoted Portfolio$^+$, and 64 cores. Here, the impact of objective probing is even larger than for the `SSS` solver. Interestingly, the difference is small on 8 cores, and grows larger when using more parallel workers, which may be a hint that the normal portfolio solver does not achieve sufficient diversification when using many cores. Furthermore, the median speedup on hard benchmarks is even higher than one obtained by the `SSS` solver.

8 A Hybrid Solver

When comparing the results of the portfolio solver with those of the `SSS` solver, it becomes obvious that these approaches work differently well on different problems. As can be seen in the Figures 21 and 22, `SSS` tends to perform better on

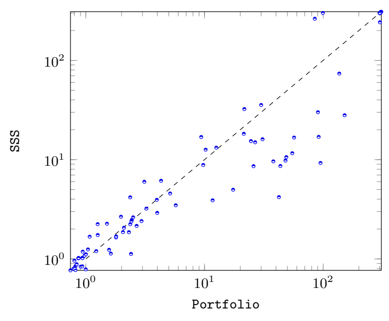

Fig. 21. `suite`: comparison between `SSS` and `Portfolio`, 64 cores

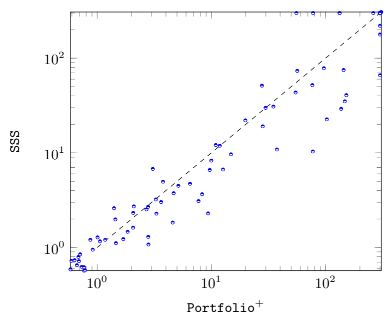

Fig. 22. `suite`: comparison between `SSS` and `Portfolio`, 8 cores

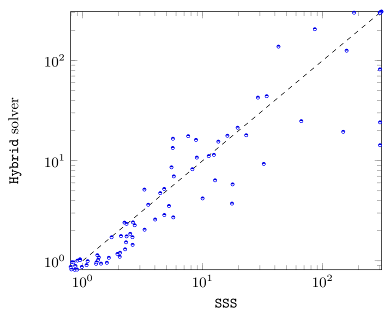

Fig. 23. `suite`: comparison between `Hybrid` and `SSS` solver, 64 cores

#CPUs	all		hard	
	avg	median	avg	median
4	4.3	3.3	18.4	6.3
8	6.3	4.7	38.2	19.9
16	9.6	6.4	79.9	43
32	11.7	8.8	116	62.4
64	15.7	16	196	140

Fig. 24. `suite`: speedups for `Hybrid` when searching good solutions

average, both on 8 and 64 cores. Nevertheless, the `SSS` solver times out on some instances that can be solved by the portfolio solver, and vice versa. It appears therefore promising to combine both approaches to a (meta-)portfolio, which combines `SSS`, initial guesses on the objective value and a SAT-like portfolio solver. We therefore change the behavior of the `SSS`$^+$-solver as follows. Workers from the second group, which finish working on the respective guessed objective values, continue working as portfolio solvers rather than joining the `SSS` solvers. Thus, they are capable of searching the whole search space instead of being fixed on one subspace, which maintains the strength of the highly agile VSIDS-based branching. We denote this as `Hybrid`. Interestingly, this is especially advantageous when using just a few cores. Here, the number of solved instances is increased remarkably. The search for good solutions is improved significantly: Using 64 cores, the median of speedups increases from 13.8 to 16 on all instances, and from 107 to 140 on the hard ones.

9 Conclusion

We presented results of different approaches to parallelize the LCG solver Chuffed. A portfolio approach performs astonishingly well, especially when trying to find good solutions rather than proving optimality. Here, an approach based on search space splitting is more successful, although is does not scale as smoothly as classical CP solvers. To avoid redundant work and hence gain better speedups, it is important to communicate some information between the parallel solvers. The most important information is the best incumbent objective value, whereas the impact of exchanging longer clauses is limited.

A hybrid solver which combines probing the objective value, portfolio solving and search space splitting yields significant speedups on a wide range of benchmarks. When trying to find better results than the sequential version of Chuffed, the speedup obtained is significantly superlinear.

On the contrary, the speedups on unsatisfiable instances, e.g. when proving optimality, are sublinear, which matches results from parallel SAT solving.

Acknowledgements

The authors would like to thank Graeme Gange for the fruitful discussions, and Prof. Dirk Nowotka for providing the computational resources for the experiments conducted in this paper.

References

1. Amadini, R., Gabbrielli, M., Mauro, J.: A multicore tool for constraint solving. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 232–238. AAAI Press (2015)
2. Amadini, R., Stuckey, P.J.: Sequential time splitting and bounds communication for a portfolio of optimization solvers. In: O'Sullivan [21], pp. 108–124
3. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Boutilier [4], pp. 443–448
4. Boutilier, C. (ed.): IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009 (2009)
5. Chu, G.: Improving Combinatorial Optimization. Ph.D. thesis, University of Melbourne (2011)
6. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Gent [10], pp. 226–241
7. Ehlers, T., Nowotka, D., Sieweck, P.: Communication in massively-parallel SAT solving. In: 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014. pp. 709–716. IEEE Computer Society (2014)
8. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In: Gent [10], pp. 352–366
9. Garcia de la Banda, M., Stuckey, P.J., Hentenryck, P.V., Wallace, M.: The future of optimization technology. Constraints 19(2), 126–138 (2014)

10. Gent, I.P. (ed.): Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings, LNCS, vol. 5732. Springer (2009)
11. Gent, I.P., Jefferson, C., Miguel, I., Moore, N., Nightingale, P., Prosser, P., Unsworth, C.: A preliminary review of literature on parallel constraint solving. In: Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving (2011)
12. Hamadi, Y., Jabbour, S., Sais, L.: Control-based clause sharing in parallel SAT solving. In: Boutilier [4], pp. 499–504
13. Katsirelos, G., Sabharwal, A., Samulowitz, H., Simon, L.: Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In: desJardins, M., Littman, M.L. (eds.) Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA. AAAI Press (2013)
14. Lin, Y., Kumar, V.: Performance of and-parallel execution of logic programs on a shared-memory multiprocessor. In: FGCS. pp. 851–860 (1988)
15. Machado, R., Pedro, V., Abreu, S.: On the scalability of constraint programming on hierarchical multiprocessor systems. In: 42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013. pp. 530–535. IEEE Computer Society (2013)
16. Martins, R., Manquinho, V., Lynce, I.: Parallel Search for Boolean Optimization. In: RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (2011)
17. Moisan, T., Quimper, C., Gaudreault, J.: Parallel depth-bounded discrepancy search. In: Simonis, H. (ed.) Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings. LNCS, vol. 8451, pp. 377–393. Springer (2014)
18. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001)
19. Oh, C.: Between SAT and UNSAT: the fundamental difference in CDCL SAT. In: Heule, M., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. LNCS, vol. 9340, pp. 307–323. Springer (2015)
20. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
21. O'Sullivan, B. (ed.): Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings, LNCS, vol. 8656. Springer (2014)
22. Rao, V., Kumar, V.: Superlinear speedup in parallel state-space search. In: Nori, K., Kumar, S. (eds.) Foundations of Software Technology and Theoretical Computer Science, LNCS, vol. 338, pp. 161–174. Springer Berlin Heidelberg (1988)
23. Régin, J., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings. LNCS, vol. 8124, pp. 596–610. Springer (2013)
24. Régin, J., Rezgui, M., Malapert, A.: Improvement of the embarrassingly parallel search for data centers. In: O'Sullivan [21], pp. 622–635
25. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.: Search combinators. Constraints 18(2), 269–305 (2013)

26. Schulte, C.: Parallel search made simple. Technical Report TRA9/00, School of Computing, National University of Singapore, 55 Science Drive 2, Singapore 117599 (Sep 2000), to appear.
27. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. Transactions on Programming Languages and Systems 31(1), 2:1–2:43 (Dec 2008)
28. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. Constraints 16(3), 250–282 (2011)