Enhancing State-of-the-Art Parallel SAT Solvers Through Optimized Sharing Policies

Vincent Vallade[1,*,†], Julien Sopena[1,†] and Souheib Baarir[2,†]

[1]Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
[2]EPITA, LRE, F-94270, Le Kremlin-Bicêtre, France

Abstract

In the context of parallel SAT solving, efficient information sharing plays a pivotal role in improving the performance. Traditionally, this information comprises the clauses learned by the underlying sequential solvers of the parallel strategy. This paper investigates the integration of existing sharing strategies and proven heuristics into `parkissat-rs`, a parallel solver using the `kissat` sequential engine. The investigation focuses on evaluating the trade-offs between dedicating a core to solving the formula and optimizing the sharing mechanism. We evaluate different configurations of `parkissat-rs`, using additional threads for sharing, a dynamic threshold for clauses filtering, a dedicated thread for strengthening shared clauses, and a mechanism to prevent the sharing of duplicate learned clauses. We also performed a scaling study on a subset of the parameters. These extensive experiments conducted on the SAT 2022 main track benchmark demonstrate the effectiveness of the derived solvers, especially on UNSAT instances.

Keywords

Parallel SAT Solving, Sharing Strategy, Clause Strengthening

1. Introduction

Modern Conflict Driven Clause Learning (CDCL) SAT solvers have proven successful in solving various real-world problems, including those arising from hardware and software verification. As many-core machines become ubiquitous, these solvers have been adapted to operate in parallel. In this parallel context, efficient information sharing plays a crucial role in improving performance. Typically, this is achieved by exchanging sets of newly derived lemmas among the participants of the parallel solving strategy, i.e., the underlying sequential solvers.

The goal of this paper is to identify a combination of proven techniques that can enhance the quality of the sharing mechanism, leading to significant improvements in parallel SAT solving performance. These techniques have been demonstrated in the `p-mcomsps` solver [1], which exhibited excellent performance on UNSAT instances in previous SAT competitions. On the 2022 SAT competition[1], the solver `parkissat-rs`, an instance of the Painless framework

14th International Workshop on Pragmatics of SAT (PoS 2023)
*Corresponding author.
†These authors contributed equally.
vincent.vallade@lip6.fr (V. Vallade); julien.sopena@lip6.fr (J. Sopena); souheib.baarir@epita.fr (S. Baarir)
0009-0001-8821-0357 (V. Vallade); 0000-0001-8140-0273 (S. Baarir)

[1]https://satcompetition.github.io/2022/index.html

Solvers	CDCL	TIME	SAT
VBS	-	16H13	167
P-2G	30	18H17	167
P-1G	31	21H23	164

Solvers	CDCL	TIME	UNSAT
VBS	-	53H33	165
P-1G	31	54H42	165
P-2G	30	54H46	164

Table 1
Performance evaluation of the -XG options.

[2] utilizing `kissat` as sequential engine [3], achieved outstanding results on both SAT and UNSAT instances. However, the sharing mechanism configuration used in `parkissat-rs` differs significantly from that of `p-mcomsps`. Therefore, we aim to investigate if the sharing policies of `p-mcomsps` can be incorporated into `parkissat-rs` [4] to further enhance its performance.

Over the years, `p-mcomsps`'s sharing mechanism has undergone incremental improvements with multiple options. Some options involve simple decision heuristics, while others require dedicated execution units (threads). Since we intend to use a solver with different sequential algorithms and diversification mechanism, we need to evaluate the trade-offs between dedicating a core to solving the formula versus optimizing sharing.

We evaluate various configurations of `parkissat-rs` with the following options: (i) employing additional threads to handle sharing, thereby increasing the bandwidth of learned clauses; (ii) utilizing a dynamic threshold for the Literal Block Distance measure; (iii) introducing a dedicated thread for strengthening the exchanged clauses; (iv) implementing a mechanism to prevent the sharing of duplicate learned clauses.

We show that option (i) alone is sufficient to significantly improve performance on SAT instances. However, for better results on UNSAT instances, option (ii) must be activated and it combines well with options (iii) and (iv), albeit without spectacular improvements.

Furthermore, we present some insights into the results through a scaling experiment performed on machines with 32, 48, and 64 cores. We demonstrate that the solving times of SAT instances decrease as the number of cores increases, while the solving times of the UNSAT instance do not decrease beyond 48 cores.

The structure of the article is as follows: Section 2 describes the architecture and specificities of the parallel solver `parkissat-rs`. Section 3 outlines the framework and methodology employed in the study. The options integrated into the sharing policy are discussed in Section 4. Section 5 discusses some scaling results and Section 6 presents the concluding remarks of the paper.

2. Parkissat-rs

In this section, we present `parkissat-rs` (see Figure 1), a state-of-the-art parallel (portfolio) solver that utilizes shared memory-based communication among the underlying sequential solvers. As the winner of the parallel SAT competition in 2022 by a significant margin, this customized portfolio of `kissat` solvers has proven to be highly effective. The diversification mechanism of `parkissat-rs` randomizes the branching order of variables for each solver in the portfolio, supplemented with the options provided by `kissat`, the underlying solver.

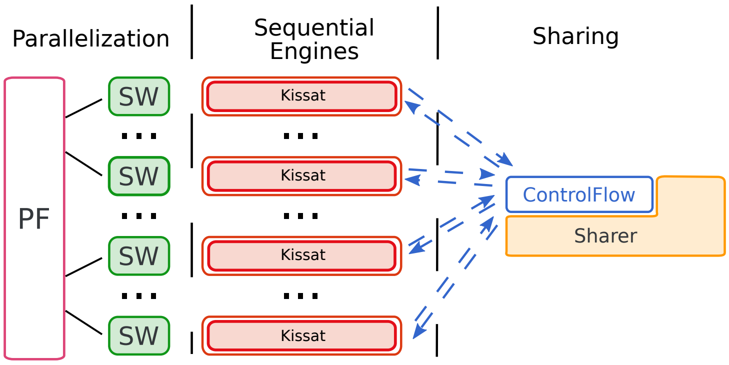

Figure 1: Architecture of `parkissat-rs`, three main components define: the *Parallelization* (here a portfolio) and its `SequentialWorker` which represent the execution of an algorithm, the *Sequential Engines* which are adapters for plugging the implementation of an algorithm (here `kissat`) into the framework and finally *Sharing* which defines a sharing policy.

Additionally, several preprocessing and in-processing methods have been developed to guide the solver at the beginning or during the resolution process.

Furthermore, `parkissat-rs` implements a policy for sharing learned clauses. Initially, each solver places all the clauses it has learned with a Literal Block Distance (LBD) value less than or equal to 2 into an export buffer. Then, a dedicated sharing thread called `Sharer` retrieves the clauses from all export buffers and distributes them to the import buffers of all solvers. The sharing process is performed asynchronously, thereby avoiding contention on locks since each buffer is only accessed by two threads: the thread responsible for sharing the clauses and the `Sharer` thread. The sharing thread performs this operation every 0.5 seconds, and to control the bandwidth of shared data, it only shares 1500 literals per sharing round.

In Section 4, we will incrementally introduce various mechanisms to enhance the sharing policy of the aforementioned architecture. We will analyze the impact of these mechanisms on the performance of the platform.

3. Experimental Setup and Evaluation Methodology

Here, we present the methodology employed to evaluate the potential combinations of options, as well as provide a detailed overview of the experimental platform.

Hardware The experiments were conducted on a cluster consisting of Intel Xeon Silver 4216 processors, equipped with 32 cores and 384 GB of RAM. The solvers were executed with a timeout of 5000 seconds and a memory limit of 256 GB.

The benchmark We executed our solver on the main track of the SAT 2022 competition, which comprises a minimum of 171 SAT instances and 187 UNSAT instances, as per the competition results. Consequently, we divided the results into two separate tables: one for SAT instances and another for UNSAT instances. However, we excluded the 42 instances with UNKNOWN results, both as reported by the competition and observed in our experiments, from the tables.

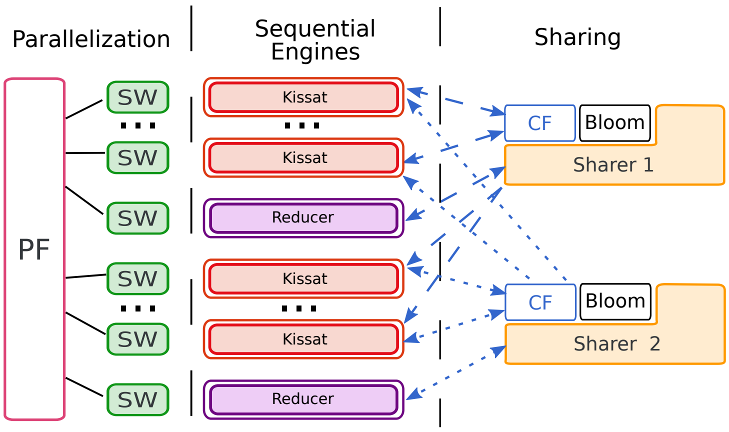

Figure 2: Final architecture after adding all the options (`P-2G-horde-str-dup`).

Performance evaluation Our objective is to compare the performance of each solver with the *Virtual Best Solver (VBS)*, which is composed of the best results achieved by any solver for each instance. We assess the solvers based on their execution time and the number of instances solved. In each table, we present the performance of a solver alongside the number of cores dedicated to the CDCL algorithm.

Since we aim to utilize all available physical cores on our test machines and certain options require a dedicated thread, some configurations involve sacrificing a physical core used for CDCL (as in the original version, `parkissat-rs`) in favor of the specific option.

The tables include the following data:

- The "CDCL" column indicates the number of CDCL cores utilized in the portfolio.
- The "Total Runtime" represents the cumulative runtime of the solver.[2]
- The "Instances Solved" denotes the number of instances successfully solved.

4. Improvements of the sharing policy

In this section, we introduce various enhancements to the sharing mechanism of `parkissat-rs` and examine their performances. Each improvement is individually studied, as well as in combination with others, as they can have both quantitative and qualitative effects on clause sharing. Additionally, these mechanisms come with associated resource costs, making it crucial to investigate their potential side effects on each other.

To streamline the presentation of the results, we have opted to incrementally incorporate each studied mechanism. Towards the end of the section, we will present all the possible combinations. The resulting solver is named `P-[options]` for clarity and reference.

[2] A non-solved instance is given a runtime of 5000.

4.1. XG: Increasing sharing throughput using multiple production groups

The first option we introduce to the sharing policy is the concept of *multiple production groups*. A production group comprises a set of solvers as producers, all solvers in the portfolio as consumers, and a dedicated `Sharer` thread running on a specific physical core.

The original implementation of `parkissat-rs` utilizes one production group (referred to as `P-1G` in the tables). To enhance the throughput of shared clauses, we incorporate a second production group: half of the solvers in the portfolio act as producers for one group, while the other half act as producers for the other group. This updated configuration is referred to as `P-2G`.

While the proposed mechanism is fully customizable and allows for additional groups, preliminary tests on our test machine have shown performance degradation when using more than 2 groups. The increase in the number of physical cores dedicated to different threads becomes too significant. Figure 2 shows the final architecture after incorporating all the options (`P-2G-horde-str-dup`).

Table 1 compares `P-1G` and `P-2G`. We observe that using two sharing groups yields improved performance on SAT instances, with three additional instances solved. However, there is no performance improvement for UNSAT instances. Although `P-2G` loses one instance, the runtime is nearly equivalent, indicating that `P-1G` is not significantly faster.

The second `Sharer` enables faster sharing of clauses, leading to a more rapid utilization of the shared clauses. However, since the strategy only shares clauses with an LBD value less than or equal to two, the impact of these shared clauses is limited.

4.2. Horde: A Heuristic to Select Clauses to Share

The `-horde` option enables the `Sharer` to dynamically adjust the LBD threshold for individual producers based on an estimation of whether they are sending too few or too many clauses. This mechanism allows for better control over the sharing of clauses. It is worth noting that this sharing strategy is an improvement of the one developed in [5].

Table 2 presents the performance results of using the `-horde` option with `P-1G` and `P-2G`. Although this option appears to slightly degrade the performance for SAT instances, specifically with `P-1G-horde` losing two instances compared to `P-1G`, a closer examination of the runtime indicates that `P-1G-horde` overall achieved faster execution.

On the other hand, the `-horde` option delivers significant performance gains for UNSAT instances, with both solvers benefiting from up to five additional solved instances.

The key insight from this experiment is that clauses with an LBD greater than two prove to be highly valuable for solving UNSAT instances. Notably, the `-horde` option performs on par with the VBS in solving UNSAT instances.

4.3. STR: Asynchronous clause strengthening

The `-str` option activates the *reducer* component, which implements an asynchronous clause strengthening algorithm. The purpose of this option is to enhance the unit propagation (UP) procedure by reducing the size of received learned clauses. The reducer operates within a production group and attempts to minimize the size of the clauses received from the solvers

Solvers	CDCL	TIME	SAT
VBS	-	13H26	169
P-2G	30	18H17	167
P-2G-horde	30	18H58	167
P-1G	31	21H23	164
P-1G-horde	31	21H28	162

Solvers	CDCL	TIME	UNSAT
VBS	-	44H44	169
P-1G-horde	31	45H42	169
P-2G-horde	30	46H03	169
P-1G	31	54H42	165
P-2G	30	54H46	164

Table 2
Performance evaluation of the `-horde` option.

Solvers	CDCL	TIME	SAT
VBS	-	12H12	169
P-2G	30	18H17	167
P-2G-horde	30	18H58	167
P-1G-horde-str	30	19H51	165
P-2G-str	28	20H19	164
P-1G	31	21H23	164
P-2G-horde-str	28	21H28	164
P-1G-str	30	22H03	163
P-1G-horde	31	22H38	162

Solvers	CDCL	TIME	UNSAT
VBS	-	42H39	169
P-2G-horde-str	28	45H13	169
P-1G-horde-str	30	45H14	169
P-1G-horde	31	45H42	169
P-2G-horde	30	46H03	169
P-2G-str	28	48H43	169
P-1G-str	30	53H07	164
P-1G	31	54H42	165
P-2G	30	54H46	164

Table 3
Performance evaluation of the `-str` option.

within the group. Successfully reduced clauses are then sent back to the solvers. Since the reducer is computationally intensive, a dedicated core is allocated for its execution.

The theoretical basis of this strengthening technique is presented in [6], and here we provide the technical details. The reducer relies on a standard CDCL algorithm. It takes a clause as input and outputs the same clause, potentially reduced by some or all of its literals. If the reducer derives the empty clause during the process, it indicates that the formula is unsatisfiable (UNSAT). The algorithm operates as follows:

1. Iteratively assign the false value to each variable in the clause until a conflict is reached or all literals have been successfully assigned.
2. At each iteration, select a literal whose complement is not involved in the current assignment. This ensures that the input clause is stripped of literals that are redundant based on the rest of the clause.
3. Perform UP after selecting each literal. If no conflicts are found, add the literal to the output clause and add its negation to the current assignment set.
4. When a conflict is reached, execute a sequence of backjumps, UP, and conflict analyses until the conflict is resolved or the current assignment set is empty.
5. During this phase, the algorithm can learn new clauses. When it reaches an area without conflicts while the assignment set is not empty, it returns a new clause.

A previous evaluation demonstrated that a similar architecture reduced the size of one third of received clauses by a quarter [7].

Table 3 presents the performance results of the `-str` option. We can observe that the integration of this option yields consistent improvements for UNSAT instances across all the

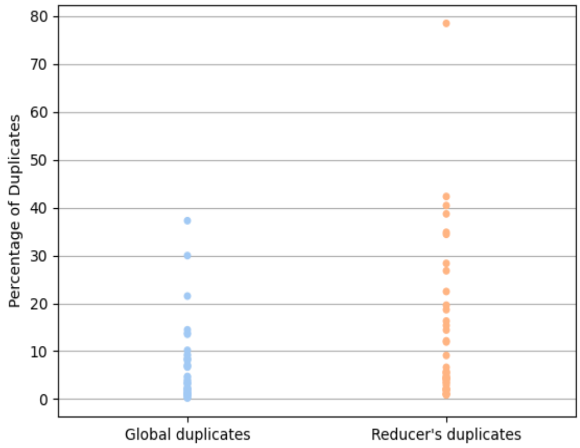

Figure 3: Percentage of duplicate clauses coming from the CDCL core and the reducer core

previously defined configurations, either in terms of running time or the number of solved instances. Notably, the `P-2G-str` configuration outperforms the `P-2G` configuration in terms of the number of solved instances.

However, for SAT instances, the impact of the `-str` option is generally negative, as three out of four configurations experience performance degradation. The `P-1G-horde-str` configuration appears to be an anomaly, which could be attributed to the inherent randomness affecting the solver's behavior.

In summary, assigning a core that was originally dedicated to executing CDCL to the execution of the reducer component proves to be highly beneficial for solving UNSAT instances but has an adverse effect on SAT instances. These findings align with the results of experiments conducted in [6], the original context in which the strengthening algorithm was developed.

4.4. DUP: Preventing the Sharing of Duplicates

In a parallel context, it is common to encounter many identical clauses that are learned by different CDCL solvers. Sharing these duplicates among solvers can be detrimental to the parallel solver's performance.

To address this issue, it is beneficial to develop a mechanism that detects and rejects the sharing of duplicated clauses. However, such a mechanism comes with a cost that can potentially impact the sharer's performance. Therefore, it is crucial to ensure that the number of duplicate clauses is significant enough to justify the trade-off.

The study depicted in Figure 3 examines the percentage of duplicates originating from CDCL solvers (Global duplicates) and from the reducer (after the strengthening phase). Each dot in the plot represents the percentage for a specific instance[3].

---

[3]The benchmark used here is from the 2021 SAT competition

Solvers	CDCL	TIME	SAT
VBS	-	09H53	170
P-2G	30	18H17	167
P-2G-horde	30	18H58	167
P-2G-horde-str-dup	28	19H38	164
P-1G-horde-str	30	19H51	165
P-2G-str	28	20H19	164
P-1G	31	21H23	164
P-1G-horde	31	21H28	162
P-1G-str-dup	30	21H37	164
P-1G-horde-dup	31	21H52	163
P-1G-str	30	22H03	163
P-1G-dup	30	22H31	164
P-2G-horde-dup	30	22H31	163
P-2G-horde-str	28	22H38	164
P-2G-dup	30	23H48	162
P-1G-horde-str-dup	30	23H53	161

Solvers	CDDL	TIME	UNSAT
VBS	-	42H06	169
P-1G-horde-dup	31	45H05	169
P-2G-horde-str	28	45H13	169
P-1G-horde-str	30	45H14	169
P-2G-horde-dup	30	45H31	169
P-1G-horde	31	45H42	169
P-2G-horde-str-dup	28	45H47	169
P-1G-horde-str-dup	30	45H56	168
P-2G-horde	30	46H03	169
P-2G-str	28	48H43	169
P-1G-str	30	53H07	164
P-1G-dup	31	53H35	165
P-1G-str-dup	30	53H36	164
P-2G-dup	30	54H29	165
P-1G	31	54H42	165
P-2G	30	54H46	164

Table 4
Performance evaluation of -dup option

The study reveals that CDCL solvers generate between 0% and 10% duplicates. On the other hand, the reducer produces a substantial number of duplicates across a larger number of benchmark instances. This analysis motivates the introduction of a duplicate removal mechanism.

Implementing such a mechanism requires tracking the already shared clauses. However, storing all clauses perfectly would be memory-intensive. Therefore, we chose to implement a space-efficient probabilistic approach using Bloom filters [8]. The sharer is enhanced with a Bloom filter to prevent the sharing of duplicates within its production group. It is worth noting that similar filters have been utilized in [5, 9]. In our configuration, this mechanism is activated by using the -dup option.

The results obtained using the -dup option are shown in Table 4. It can be observed that the outcomes are somewhat mitigated. Our intuition suggests that although a Bloom filter offers efficiency, the process of verifying duplicate clauses in each round of sharing can still impede sharing. This trade-off does not appear to be worthwhile, especially in the context of this specific benchmark with the chosen solver.

5. Scaling study

Thus far, our research has focused on analyzing the impact of different mechanisms on a configuration consisting of 32 physical cores, which reflects the characteristics of machines used in SAT competitions over the past few years. However, the issue of scalability of these algorithms has emerged as a significant concern.

To address this concern, we conducted a new study using two additional architectures: one with 48 cores and another with 64 cores. Within this study, we concentrated on

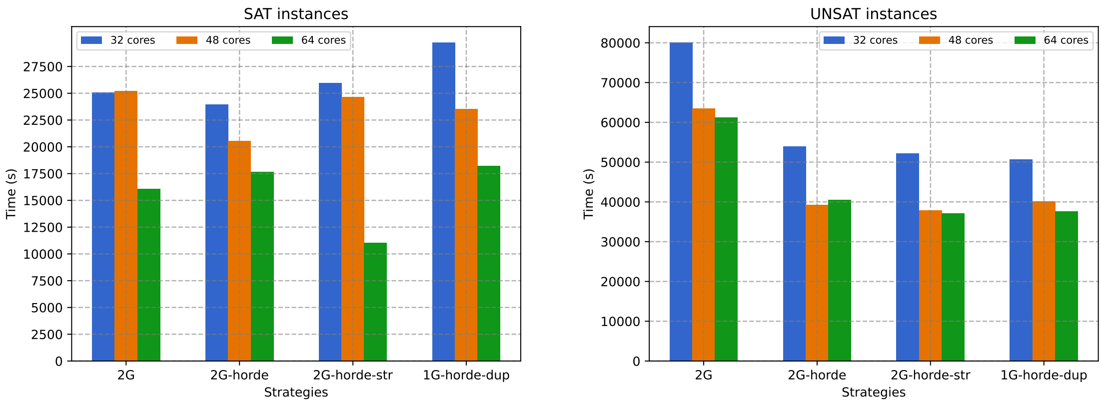

Figure 4: Scaling experiment

the best-performing combinations identified in the 32-core study: namely, `2G`, `2G-horde`, `2G-horde-str`, and `1G-horde-dup`. To assess their performance, we employed a benchmark comprising 40 instances that were solved by at least one configuration utilizing the 32-core architecture, with each instance having a maximum allowed solving time. The results of this study are presented in Figure 4.

Overall, our findings indicate that the relative performance of the different combinations, with the exception of one case, remains consistent as the number of cores increases, regardless of whether the problem is SAT or UNSAT. However, we observed an intriguing phenomenon: beyond 48 cores, scalability notably diminishes for UNSAT problems, irrespective of the algorithm employed. This can be attributed to the limitations of the sharing mechanisms, as the additional threads do not effectively reduce the search space explored by other threads. Conversely, SAT problems exhibit different behavior, as increasing the number of threads enhances the probability of swiftly finding a solution.

Furthermore, alongside these general results that support prior research, it is worth highlighting the remarkable performance of 2G-horde-str. While it maintains its superiority for solving UNSAT problems, it emerges as the most efficient algorithm for SAT problems in the 64-core configuration. This achievement can be attributed to the substantial impact of strengthened clauses, which enable the early pruning of irrelevant branches in the decision tree, thereby accelerating convergence towards a solution.

6. Conclusion

In this paper, we have combined established techniques with new algorithms to enhance the sharing mechanism in the parallel solver, `parkissat-rs`.

Specifically, compared to the original implementation, we introduced production groups, the horde sharing strategy, strengthening, and duplicate removal mechanisms. The complete solver with all the options is depicted in Figure 2.

We conducted experiments and compared the performance of each implemented add-on on the benchmarks of the 2022 SAT competition. This allowed us to draw conclusions regarding the individual and combined effects of each technique. For SAT instances, the most effective implementation is P-2G. This implies that the original implementation is already efficient and requires minimal changes. The division of solvers within production groups significantly accelerates sharing.

For UNSAT instances, all implementations utilizing the horde sharing strategy are highly efficient. This mechanism complements strengthening and duplicate removal when used separately, but combining all three does not yield improved results.

These conclusions hold for multi-core machines with up to 100 cores. Further exploration of these strategies in many-core machines with more than 100 cores would be fascinating. In such scenarios, the cost of CPU-intensive mechanisms could be offset by utilizing a larger number of solvers, achieving better resource utilization.

Furthermore, a possible heuristic avenue, not explored in this paper, is the diversification mechanism used by parallel SAT solvers to avoid redundant work between worker threads. We are interested in directly modifying the internal structure of the sequential engine by shuffling the literal order of its clause database. This modification will impact the unit propagation of the solver and may yield different results.

References

[1] V. Vallade, L. Le Frioux, R. Oanea, S. Baarir, J. Sopena, F. Kordon, S. Nejati, V. Ganesh, New concurrent and distributed painless solvers: P-mcomsps, p-mcomsps-com,p-mcomsps-mpi, and p-mcomsps-com-mpi, in: Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions, Department of Computer Science, University of Helsinki, Finland, 2021, p. 40.

[2] L. Le Frioux, S. Baarir, J. Sopena, F. Kordon, Painless: a framework for parallel sat solving, in: Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT), Springer, 2017, pp. 233–250.

[3] M. S. Cherif, D. Habet, C. Terrioux, Combining VSIDS and CHB Using Restarts in SAT, in: 27th International Conference on Principles and Practice of Constraint Programming (CP 2021), LIPIcs, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021, pp. 20:1–20:19.

[4] X. Zhang, Z. Chen, S. Cai, Parkissat: Random shuffle based and pre-processing extended parallel solvers with clause sharing, in: Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions, Department of Computer Science, University of Helsinki, Finland, 2022, p. 51.

[5] T. Balyo, P. Sanders, C. Sinz, Hordesat: A massively parallel portfolio sat solver, in: Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT), Springer, 2015, pp. 156–172.

[6] S. Wieringa, K. Heljanko, Concurrent clause strengthening, in: Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT), Springer, 2013, pp. 116–132.

[7] V. Vallade, L. Le Frioux, S. Baarir, J. Sopena, F. Kordon, On the usefulness of clause

strengthening in parallel sat solving, in: R. Lee, S. Jha, A. Mavridou, D. Giannakopoulou (Eds.), NASA Formal Methods, Springer International Publishing, Cham, 2020, pp. 222–229.

[8] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (1970) 422–426.

[9] D. Schreiber, P. Sanders, Scalable sat solving in the cloud, in: C.-M. Li, F. Manyà (Eds.), Theory and Applications of Satisfiability Testing – SAT 2021, Springer International Publishing, Cham, 2021, pp. 518–534.