Parallelizing a SAT-Based Product Configurator

Nils Merlin Ullmann
CAS Software AG, Karlsruhe, Germany

Tomáš Balyo
CAS Software AG, Karlsruhe, Germany

Michael Klein
CAS Software AG, Karlsruhe, Germany

Abstract

This paper presents how state-of-the-art parallel algorithms designed to solve the Satisfiability (SAT) problem can be applied in the domain of product configuration. During an interactive configuration process, a user selects features step-by-step to find a suitable configuration that fulfills his desires and the set of product constraints. A configuration system can be used to guide the user through the process by validating the selections and providing feedback. Each validation of a user selection is formulated as a SAT problem. Furthermore, an optimization problem is identified to find solutions with the minimum amount of changes compared to the previous configuration. Another additional constraint is deterministic computation, which is not trivial to achieve in well performing parallel SAT solvers. In the paper we propose five new deterministic parallel algorithms and experimentally compare them. Experiments show that reasonable speedups are achieved by using multiple threads over the sequential counterpart.

2012 ACM Subject Classification Applied computing → E-commerce infrastructure

Keywords and phrases Configuration, Satisfiability, Parallel

Digital Object Identifier 10.4230/LIPIcs.CP.2021.55

Related Version The paper is based on Nils Merlin Ullmann's Master's thesis.
Extended Version: `https://doi.org/10.5281/zenodo.5154040`

1 Introduction

Configuration systems [12] offer great benefits for the sales process and customers, while technical challenges rise to enable configuration models with increasingly large knowledge bases. Every valid configuration has to fulfill a set of propositional formulas. The configuration system's task is to find such an assignment for a given set of user requirements. Janota showed [25] that this problem can be expressed as a *Boolean Satisfiability Problem* (SAT) and consequently solved by a SAT-solver. Essentially, the configuration model together with the user requirements are transformed into conjunctive normal form (CNF). If the formula is satisfiable, then there exists at least one valid configuration for the underlying configuration model and the user's needs.

Parallel SAT-solvers have been mainly studied on very hard SAT problems. Configuration systems tend to have different requirements that exceed the common SAT problem. Generally, the created problem instances are smaller and less complex, due to the a step-by-step configuration process. However, new problems are introduced. Firstly, in case that the customer's latest selection in the interactive process combined with the current configuration are unsatisfiable, the configuration system should return an alternative solution that minimizes the number of changes with regard to the current configuration. This problem can be modeled as an optimization problem, for example as a *Maximum Satisfiability Problem* (MaxSAT) or *Minimum-Cost Satisfiability Problem* (MinCostSAT). The underlying cost function evaluates

the changes of every assignment compared to the current configuration. Additionally a configuration system requires fully deterministic behavior and very low response time (in the order of hundreds of milliseconds).

The aim of this work is to report on how state-of-the-art parallel SAT solving techniques can be adapted and used in a commercial product configurator. Our main contribution is the development of parallel algorithms for problem instances created during interactive configuration processes [1]. These instances are described by calculating the optimal valid solutions for a given user change, start-configuration, and Boolean formula. Several parallel algorithms are presented that fulfill the completeness and optimality criteria for assignments required by configuration systems. Experimental results show that the presented approaches can achieve significant improvements in response time by using multiple processor cores. Furthermore, the approaches fulfill the completeness, optimality, and determinism requirements. Especially the latter has not been widely researched in this domain, because many applications do not rely on reproducible results.

2 Related Work

SAT and two of its extensions (MaxSAT and MinCostSAT) have been applied to the field of product configuration [46, 47]. Methods were presented to check the general consistency of a product data base, by converting it into formulas of propositional logic. This was done in a real product configuration system for the automotive industry. A broader discussion of the applicability of SAT solvers for configuration systems is given in [25], where scalability is emphasized as a potential benefit. Early research on concepts focusing on the configuration process has been presented by Sabin and Weigel in [43]. A good overview of the *interactive configuration* process, in which a user specifies his requirements step-by-step with validation and feedback in between, is given in [27]. Different solving techniques for this process have been proposed. Batory and Freuder et al. present in [3] and [13] algorithms for a lazy approach. In this context lazy means that no precompilation is required, because all computations are performed during the configuration process. Non-lazy approaches mainly use binary-decision-diagrams (BDD) instead of SAT solvers, examples are [16] and [1]. Furthermore, Janota discusses many optimization techniques and algorithms to model the interactive configuration process lazily with the help of SAT [26]. Additionally, he elaborates on methods to improve the transparency of a configuration system. This is achieved by providing algorithms to generate comprehensible explanations using resolution trees and completing partial configurations.

Currently, most parallel SAT solvers are based on one of the two main approaches: *divide-and-conquer* (also called search space splitting) and *parallel portfolios*. Early parallel SAT solvers (PSATO [49], PSatz [30], PaSAT [45], GridSAT [7], MiraXT [34], PaMiraXT [44]) were based on the divide-and-conquer approach. The search space in these solvers is divided dynamically using *guiding-paths*. Another approach is to divide the search space statically at the beginning of the search using look-ahead techniques [22]. This paradigm is called

[1] We adapted already existing algorithms, however, they are designed for SAT solving, which still has several differences to our problem (decision vs. optimization problem, non-deterministic vs. deterministic behavior, focus on large difficult problems vs. real time response, non-interactive vs. interactive usage). Therefore our main contribution is the non-trivial adaptation of the SAT algorithms to product configuration. The second contribution is the evaluation of these algorithms on real industrial configuration problems (and some random problems) and identifying their strengths and weaknesses in that context.

cube-and-conquer. It is a two-phase approach that partitions the original problem into many subproblems (cubes) which are subsequently solved in parallel [23, 5]. Parallel portfolios, popularized by Hamadi et al. with the solver ManySAT [18], differ by starting several SAT solvers on the same formula but with different parameter settings in parallel. The solvers compete to find a solution to the input problem and terminate as soon as one has been found. More recent examples of portfolio solvers are Plingeling [4] or HordeSat [2]. Most parallel SAT solvers introduce non-deterministic behavior, which is not acceptable in some applications, such as ours for example. This problem has been acknowledged by Hamadi et al. [17], who proposed the first deterministic parallel SAT solver based on ManySAT. A recent result [40] shows that comparable performance to non-deterministic parallel SAT solvers is achievable by using techniques such as delayed clause exchange and accurate estimation of execution time of clause exchange intervals.

This paper is focused on the combination of SAT and optimization problems. Hence, we also discuss parallel best-first search algorithms. Approaches for parallel A* can be separated into two major categories, depending on the management of the OPEN list. A centralized parallel A* [24, 41] works on a shared central OPEN list [41]. To remove potential bottlenecks on the shared OPEN list, algorithms that use the so called decentralized approach have been developed. The algorithm PRA* (Parallel Retracting A*) assigns an OPEN list to each processor [10]. Every generated node is mapped to a processor using a hash function. Following work mainly concentrated on developing sophisticated hash functions to reduce overhead [31, 29].

3 Preliminaries

A *Boolean variable* has two possible values: *True* and *False*. A *literal* is a Boolean variable (positive literal) or a negation of a Boolean variable (negative literal). A *clause* is a disjunction ($\vee$) of literals and, finally, a CNF formula (or just formula) is a conjuction of clauses. A clause with only one literal is called a *unit clause*. A positive (resp. negative) literal is satisfied if the corresponding variable is assigned the value *True* (resp. *False*) A clause is satisfied, if at least one of its literals is satisfied and the formula is satisfied, if all its clauses are satisfied.

The satisfiability (SAT) problem is to determine whether a given formula has a satisfying assignment, and if so, also find it. Most complete SAT solvers are based on the DPLL algorithm [8] and its extension the CDCL algorithm [38, 39].

SAT only searches for an assignment that satisfies the Boolean formula. Optimality with regard to the assignment is not considered. Two approaches that introduce an optimization function are the Maximum Satisfiablity Problem (MaxSAT) and the Minimum-Cost Satisfiability Problem (MinCostSAT). Both problems extend SAT by incorporating a cost function that evaluates assignments. The better known MaxSAT problem is to find an assignment that maximizes the number of satisfied clauses of a Boolean formula [35]. In the MinCostSAT problem we assign a non-negative cost to each variable to quantify an assignment. The problem is to find a variable assignment that satisfies F and minimizes the total cost of the variables set to *True*. The transformation between MaxSAT and MinCostSAT problems can be performed by adding auxiliary variables/clauses to the respective formulas [36].

The DPLL/CDCL algorithm can be extended to solve MinCostSAT instances. The cost for every variable assignment are accumulated during unit propagation. At every branching point a decision variable is chosen, partial assignments are calculated, and the assignment with the lowest costs is used. This amounts to a recursive branch-and-bound algorithm [14, 36].

A configuration task is a triplet (V, D, C), where $V = \{v_1, v_2 \ldots, v_n\}$ is a finite set of domain (feature) variables and $D = \{dom(v_1), dom(v_2), \ldots, dom(v_n)\}$ represent the set of corresponding finite variable domains. Furthermore, $C = P_{KB} \cup C_R$ represent constraints, with P_{KB} being the product knowledge base (configuration model) and C_R a set of user requirements. The *solution* to a configuration task is called a *configuration*. A configuration is an instantiation (assignment) $I = \{v_1 = i_1, v_2 = i_2, \ldots, v_n = i_n\}$, with each i_j being one of the elements of $dom(v_j)$. A configuration is called valid, if it is *complete* (every variable is assigned with a value) and *consistent* with all constraints [11]. Janota discusses in [25] whether SAT solvers can be used effectively in a configuration system (*configurator*). Janota shows that every configuration task can be translated into a Boolean formula. Consequently, by solving the Boolean formula in CNF by using a SAT solver, the initial configuration task is solved as well.

4 Problem definition

The term *configuration* can be interpreted as an assignment for an underlying Boolean formula F_{cnf}. During the interactive configuration process, a user selects or deselects attributes step-by-step to add or remove them from a configuration. The attributes are translated to corresponding Boolean variables of F_{cnf}, thus every attribute is also an atomic proposition. A selection expresses that the attribute must be part of the current configuration. Any selection or deselection can be reverted throughout the configuration process. A configuration is considered valid if it is consistent with all constraints (C), but not every variable (V) needs to be assigned. The user may start with an empty configuration which is progressing through user selections and deselections until it is complete. With respect to the underlying Boolean formula, an empty configuration contains every literal as a negative one. In the following, a user selection or deselection is also called a *user wish*.

During the configuration process, each user wish δ causes a *configuration step*. The step calculates a new valid configuration (solution) s using an existing assignment β (called *start-configuration*), by applying the user wish. A start-configuration describes the valid preexisting assignment for a configuration step. The user wish represents the desired change that should be applied to the existing configuration. Therefore, a user wish δ can be described as a set of literals.

For every configuration step, the configurator ensures the validity of the resulting configuration to prevent invalid user selections. Depending on the constraints, certain user wishes violate the configuration model which are resolved by the configurator through automatically selecting or deselecting attributes. To quantify the result of a configuration step, a cost function is introduced. The costs of the changes resulting from the user wish are calculated by analyzing the difference between the start-configuration β and the resulting configuration s:

$$deltaCost(\beta, s) = \sum_{l \in s} \begin{cases} l \in \beta \rightarrow 0 \\ l \notin \beta \rightarrow c(l) \in \mathbb{N}_{\geq 0} \end{cases} \tag{1}$$

The cost function $c(l)$ must be non-negative but can be domain specific. For example, literal changes from positive to negative can be more expensive to prefer keeping literals that the user already selected in the configuration process. A change from a positive literal to a negative one expresses a deselection of an attribute for the user. Each configuration step has two concrete requirements to fulfill.

1. Every configuration step has to apply the user wish δ to the start-configuration β, expressed as $\delta \subseteq s$.

2. The resulting configuration s has to be optimal, i.e., there is no other solution s' that results in lower costs (by applying the function $deltaCosts(\beta, s')$) and introduces the user wish δ to the start-configuration β.

The second requirement extends the SAT problem by incorporating an optimization problem of finding the minimal-cost configuration for each step. The problem of calculating a configuration step is similar to the *MinCostSAT* problem. The main difference is that positive variables do not inherently increase the costs. Costs are only accumulated by changes to the start-configuration, independent of the variable's truth value. The user can define a limit r on how many solutions should be returned at most in the case that multiple optimal valid configurations exist. All found solutions are represented by the set S. Furthermore, this set of solutions must be the same when repeating the same configuration step, i.e., we require fully deterministic calculation.

▶ **Definition 1** (MinCostConf). *The minimal-cost interactive configuration task (MinCostConf) is described by the 6-tuple:*

$$(A_p, C_p, \beta, \delta, c(l), r)$$

where A_p describes the set of attributes for a product p given its feature variables. The set of constraints for a specific product is given by C_p. The dynamic components are the start-configuration β and the user wish δ. Furthermore, a non-negative cost function $c(l)$ defines the cost for attribute $l \in A_p$, when l changes with respect to β. The maximum amount of returned solutions is limited to r. A solution s is valid if all constraints C_p are fulfilled and δ is part of each solution s ($\delta \subseteq s$). An optimal solution s is a minimal-cost assignment with respect to β ($\sum_{l \in s} c(l)$, $\forall l \notin \beta$). A valid optimal solution s is called a configuration and the set of found solutions is denoted by S.
The task is to find all configurations. In case of more than r optimal configurations, the cardinality of S is limited to r. Repeatedly solving the same task must return the same set S, i.e., the process must be deterministic.

Regarding the complexity, MinCostConf (being an extension of MinCostSAT) obviously belongs to the class of NP-complete optimization problems.

5 Parallel Algorithms for Product Configuration

This Section contains the main contributions of the paper – the description of the parallel algorithms we developed for the MinCostConf problem. This paper is based on a Master Thesis [48], which contains a more detailed description of the described algorithms including pseudo-code, examples, and figures.

5.1 Baseline: Sequential A* Search for MinCostConf

The A* algorithm [19] is an extension of Dijkstra's algorithm [9]. The A* algorithm formulates its problem as a weighted directed graph and aims to find the minimal cost path from a source node to a goal node. In this process, the algorithm constructs a search tree and always expands the most promising node. Furthermore, the A* algorithm maintains an OPEN list of nodes that have not been expanded yet. The list is ordered by the cost accumulated from the root node to the current node and the heuristic estimation of the cost to reach a goal node. A second list, called CLOSED list contains all nodes that have been expanded. To decide which path to expand, the function $f(n) = g(n) + h(n)$ is minimized over all nodes,

where $g(n)$ is the path's cost from the source node to n, which is the next node on the path. Additionally, $h(n)$ estimates the cost of extending the path from node n to the goal node. The heuristic function $h(n)$ differentiates the A* algorithm from Dijkstra's algorithm. A heuristic function is called admissible if it does not overestimate the cost from node n to the goal node. For A* it holds in general, that if h is an admissible function, then A* search is optimal [19]. Consequently, utilizing this property allows us to terminate the search as soon as a solution has been found, because the estimated costs of all other nodes are larger than the actual cost of the found solution.

The A* algorithm can be combined with DPLL and applied to the MinCostConf problem, which is an optimization problem. The DPLL algorithm constructs a search tree which can be interpreted as a weighted directed graph. Every edge serves as a unit propagation of a decision literal. The edges' weights are given by the costs of the propagated literals, thus the weights are non-negative. This leads to a search tree in which the costs can only increase or remain unchanged with increasing tree depth. Finally, the goal node is the lowest cost valid assignment. In the domain of product configuration, multiple valid optimal solutions may exist, which represent potential alternatives for the user to choose from. Thus, several goal nodes may exist. In our case, the function $g(n)$ sums the costs of already assigned literals. The set L_n describes the literals that are assigned on the path from source node n_1 to node n_i. As a lower bound for the costs from n to the goal node, the heuristic function $h(n_i) = costs(U_{n_i})$ is used, where U_{n_i} is the set of unit clauses remaining in the node n_i. It accumulates the costs of all remaining unit clauses in F, that will result from the chosen literal, because their truth value is already defined but they have not been propagated yet. This obviously constitutes an admissible heuristic.

During the A* search, node evaluations are mainly based on the costs of propagated literals. Essentially, every change with regards to the start-configuration introduces costs. The start-configuration always describes the configuration prior to the current user wish. Having selected an attribute, the user wants as few changes to the prior configuration as possible, thus a cost function is utilized.

5.2 Centralized Parallel A* Search

An intuitive solution is to extend the sequential A* algorithm to perform it in a multithreaded environment. In parallel versions of the centralized A* approach, the single OPEN list is kept [24, 41]. Therefore, k threads work concurrently on a shared OPEN list. Each thread retrieves nodes from that data structure in order of their f-values, expands them, and inserts the successors of the expanded nodes (states) into the OPEN list. Re-expansions of nodes (contrary to the sequential algorithm) are possible, because a state may not have the optimal g-value when taken from the list and being expanded. However, this issue is not applicable to the DPLL algorithm, since nodes in the search tree are only reached by one specific path. This property leads to two improvements. Firstly, the g-value of a node cannot be updated by processing another node first. Secondly, duplicate detection of states is not required, because only one path leads to every state, thus two threads cannot arrive at the same state. Therefore, when using parallel A* search for the DPLL algorithm, the CLOSED list is not required. Nevertheless, the concurrent work can lead to search overhead by expanding suboptimal nodes that would not have been expanded by a sequential version of A*. Besides search overhead, expanding nodes in parallel easily leads to nondeterministic behavior. This critical problem as well as solutions are discussed thoroughly at the end of this section.

In a multithreaded environment, a coordination mechanism for all threads is required. We use the *Master-Worker* paradigm, with a single master thread and a set of worker threads. The latter can scale from 1 to k threads. The master maintains a overview of the procedure,

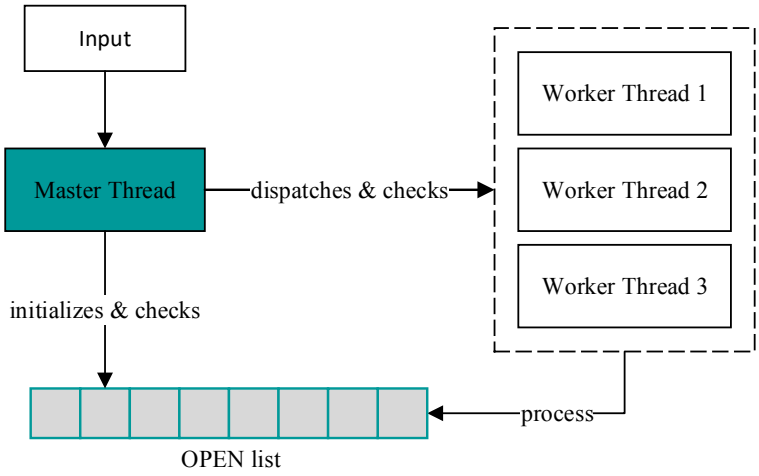

Figure 1 An explanatory thread overview of the centralized parallel A* approach using a single master thread and three worker threads (Master-Worker paradigm). All threads are accessing one central OPEN list, using a shared-memory architecture.

while the workers process tasks in parallel. The master initializes the MinCostConf instance resulting in the root node and dispatches the k threads to start processing. Subsequently the workers remove nodes from the OPEN list, which is implemented as a priority queue. Each task consists of the unit propagation of a decision literal (except for the root node) and resulting unit clauses. Communication between threads is performed by using shared data, e.g. for thread state information and all currently known optimal solutions. Access to the shared data is required for workers as well as for the master. Workers need to know the current optimal solutions to decide whether their task (node) needs to be processed or can be skipped. While all workers are processing nodes, the master checks whether the OPEN list is empty, because an empty list indicates that the search is finished. Additionally, the master checks the state of all worker threads by accessing shared data, to decide whether the search can be terminated or not, for instance in case that the optimal solution has been found. See Figure 1 for an overview diagram.

The centralized parallel A* approach uses two shared primitive variables, the Boolean $searchFinished$ and the integer $minCost$. Especially the latter is updated several times during a configuration step, because many suboptimal solutions may be found during the parallel search process. Thus, multiple threads try to retrieve and update the value of this variable simultaneously. To avoid memory inconsistency, all updates to these variables use optimistic locking in form of atomic compare-and-swap instructions. Moreover, several complex data structures are utilized. The most central data structure is the priority queue Q, representing the OPEN list by maintaining all nodes that may be expanded during the tree search. It is based on a binary heap and uses locks as a synchronization mechanism to allow parallel work of k threads on a single queue. An increasing number of threads (k) can result in high lock contention on Q. However, the assumption is that the number of operations on Q is relatively low in the domain of product configuration, due to solving smaller problem instances. Moreover, the node expansion is expensive, because it involves performing compute-intensive unit propagation which further limits the lock contention. For

search termination detection, two arrays thread-waiting states (TWS) and running-thread-costs (RTC) are used. Both enforce locking mechanisms to restrict concurrent thread access, for instance of the master thread and worker threads. While a worker thread retrieves a node from Q, the master thread has to wait until the worker thread has updated its waiting state as well as running costs. The described locking techniques are also applied in all other approaches discussed below.

5.3 Decentralized Parallel A* Search

In a decentralized parallel A* search, each thread maintains its own local OPEN list to perform the search. Initially, the root node is assembled and processed by a thread. Subsequently generated nodes are distributed among all threads to achieve good load-balancing with low idle time. Especially the load balancing strategy is a deciding factor for computation time, thus many different ideas have been developed. Kumar, Ramesh, and, Rao were among the first to utilize a distributed strategy in [33]. Their approach generates an initial amount of nodes and distributes them to all OPEN lists. The different threads expand the nodes in parallel. To avoid threads working on suboptimal parts of the search tree, since the thread is limited to its own OPEN list, they introduced a communication strategy to share nodes. The goal is to have all threads working on promising sections of the search space. The communication to distribute newly spawned nodes is performed by choosing another thread *randomly* and inserting the node to the target's local OPEN list.

To apply the decentralized parallel A* search to DPLL, each of the k threads maintains a local priority queue as an OPEN list. Moreover, a distribution strategy is required to attain good load balancing. Besides additional priority queues, the overall coordination mechanism is similar to the presented centralized parallel A* search. Therefore, a master thread and k worker threads are used. In the following, the responsibilities of the different threads are explained. Being a variation of parallel A* search, only key differences to the centralized approach are elaborated.

The master thread's function is the search initialization and termination detection. Being an extension of the centralized A* search algorithm, only a few changes have been made for the master thread's logic. For initialization, the root node is inserted into the queue of the first thread. Successive nodes are distributed by the worker threads. Having multiple OPEN lists changes the search termination detection. Firstly, we must test whether all queues in Q are empty and all threads are on a waiting state. Secondly, we test whether all remaining nodes in all OPEN lists are more expensive or at least equal-cost and r solutions have been found. Thus, the search termination detection is more complex, to accommodate the increased number of priority queues.

The worker thread only processes nodes from its local task pool to reduce lock contention. One major difference between the centralized and decentralized strategy is the handling of successor nodes. Every successor node is assigned to one specific thread, determined by a random distribution function.

Comparing properties of the decentralized and centralized parallel A*, the main advantage of the former is the reduced lock contention on the shared OPEN list. With increasing number of threads, the contention on the central data structure surges (synchronization overhead). Hence, the decentralized parallel A* search has the potential to scale better with more processing units. However, distributing nodes across multiple OPEN lists brings disadvantages as well. Firstly, communication overhead is increased, because nodes are exchanged across threads to reduce idle time and the number of suboptimal nodes expanded. Secondly, search overhead is increased compared to the centralized approach. In general, it

can be measured by comparing the expanded nodes in parallel to the number of its sequential implementation. Reason for this growth is that threads potentially work on suboptimal parts of the search tree while waiting for more promising nodes from other threads. In a centralized A* search, all threads work on one data structure, thus promising nodes are expanded first and threads often work on similar sections of the search tree.

5.4 Parallel Cube-and-Conquer

Cube-and-Conquer (C&C) is a two-step approach to solve the SAT instances. First, the problem is divided into a large number of subproblems (cubes) which are subsequently solved in parallel. Regarding thread management, the Master-Worker paradigm is used, comparable to previous parallel A* search algorithms. The master thread is entirely responsible for the first phase, thus generating the desired amount of cubes. Afterwards, the master distributes the cubes among all k available worker threads which process them independently in parallel. The generation of cubes is performed by best-first search until the required amount of cubes is found. In case the set of optimal solutions is found during this phase, the second phase is not performed.

In phase two each worker thread is assigned a list of subproblems (called *cubes*), that have to be processed. For each cube, a local priority queue Q is initialized with the cube, a partial assignment, as the root node. Subsequently, a best-first search is executed for the current cube constructing a sub-tree of the original search space. Thus, nodes are removed from the priority queue iteratively and successor nodes are generated by unit propagation and the choice of the next decision literal resulting in up to two child nodes. Furthermore, to reduce search overhead, cube termination detection is performed. If the head of the priority queue has accumulated costs that exceed the costs of any found solution then the cube can be aborted.

Considering cube-and-conquer, it is a two-phase method that is suited for hard SAT instances. For such CNF formulas, cube-and-conquer approaches can generate between thousand and a million cubes, evaluated extensively in [23] and [5]. The first phase is executed sequentially, followed by parallel processing of subproblems. Distributing cubes in phase two among all worker threads results in a relatively clear search space partitioning and thus little communication overhead. Worker threads only check the currently known minimum cost solution to decide whether a cube can be aborted. Using local information also reduces the synchronization overhead, especially by not using a global OPEN list. Overall, these advantages lead to a simpler approach with less complexity. However, the used method partitions the original search tree into sub-trees. These cubes are processed iteratively, which potentially leads to larger search overhead. The reduced communication and iterative processing of cubes can increase the time threads spend on expanding suboptimal parts of the overall search space. According to our observations, this disadvantage is enhanced in SAT instances that are not very hard, for instance a configuration step within a configuration system. Due to the underlying optimization problem (MinCostConf) and lower instance hardness, the search trees usually have a small height and width. Two reasons are that many paths can be pruned and only few conflicts are encountered. Having hard SAT instances, search trees tend to be of greater height and width, reducing potential search overhead.

In contrast to this, the parallel A* Search uses more complex search space splitting strategies, by extensively sharing data. On the one hand, regarding communication and synchronization overhead, Cube-and-conquer introduces the smallest efforts. However, this benefit leads to the drawback of higher search overhead, especially compared to the centralized A* Search approach. These properties affect the performance depending on the considered

formula's complexity. Summarized, cube-and-conquer potentially performs well on hard SAT instances, for example for Random SAT and Random 3-SAT benchmarks. Regarding industry cases, the performance is dependent on the size of the constructed search tree. Full evaluation, analysis and comparison of the different methods is presented in Chapter 6.

5.5 Parallel Portfolio Solver

Our first portfolio approach combines several instances of a sequential solver (A* base solver). Each instance has its own set of configuration settings, consisting of several parameters such as:

- The branching heuristic. Different score-based branching heuristics are used which are comparable to the One-Sided (OS) and Two-Sided (TS) Jeroslow-Wang Rules [28]. Furthermore, a second component is added as a parameter which influences the priority of choosing variables that have a positive cost (i.e. are not special variables with zero cost). Due to the underlying optimization problem, it can be beneficial to preferably branch over variables with positive cost to limit the search tree growth.
- Tie-breaking criteria of the OPEN list. We can use different strategies to prioritize nodes with the same f value within the OPEN list, such as comparing the number of unsatisfied clauses.
- Clause Learning. Clause learning is not always beneficial in MinCostConf, due the introduced overhead. Therefore some instances will utilize clause learning and others not.

With various defined parameters, a portfolio can be constructed by combining solvers with different configuration sets. The diversification's purpose is to have a portfolio with solvers that have little search space overlap to reduce search overhead and the solver's sensitivity to parameters.

Comparing parallel portfolios to divide-and-conquer approaches, several distinctions can be drawn. A major advantage is the robustness of parallel portfolios against the impact of configuration parameters that SAT solvers generally face. However, considerable disadvantages exist, especially for industrial use cases. The main problem is that the parallel portfolio does not partition the search space into distinct portions, but rather duplicates the Boolean formula for each instance. As a result, the memory consumption increases approximately proportionally to the number of cores. For industry cases and production systems, resources are limited.

5.6 Parallel Portfolio A*

An alternative strategy is to adapt the parallel A* algorithm. The main motivation is to avoid the Boolean formula duplication that is performed by the previously shown parallel portfolio. Instead, the search space is divided but multiple threads work cooperatively on the same instance. To adopt the benefit of being less sensitive to parameter tuning, each thread uses different settings but work on the same formula. Regarding parameters, changing the branching heuristic and related settings such as the "Prefer Cost Literals" can be changed for each processing unit. Hence, when a thread processes a node and has to choose the next decision literal for that path, a thread specific heuristic is utilized. Usually, these parameters have a strong impact on the search behavior with respect to the order of path expansions. This strategy can be applied to both, centralized and decentralized parallel A* search. Despite using varying parameters, no other changes are required for these algorithms.

5.7 Making the Search Deterministic

Most current parallel SAT solvers are not capable of producing stable results, due to their architectures relying on weak synchronization [17]. Nevertheless, reproducibility of a configuration process is a key requirement for a configurator.

The following criteria are used to order configurations as the foundation to achieve reproducible configuration steps: (1) cost of the configuration using the delta-cost function, (2) number of decision literals (tree-depth), (3) number of positive decision literals (left branches), and (4) hash of literals contained in the solution. During parallel search in the search tree, solutions can be found in varying order across several executions. To accommodate this instability, the termination criteria must be adapted to ensure that previously returned solutions are not skipped through early termination in another execution of the same configuration step. Therefore, two alterations are presented in the following, both aiming at ensuring deterministic behavior as well as minimizing the search overhead.

5.7.1 Tree-depth Depending Termination

The first approach exploits the number of decision variables in any obtained solution. As soon as the requested amount of alternative configurations r has been found, the maximum tree-depth (number of decision variables) j is extracted from all found solutions. This indicates the solution that is ordered last among all solutions, thus it can be used to prune unexplored nodes. Consequently, only paths within the search tree that either have lower costs than already known valid assignments or are equal-cost and have a tree-depth less or equal to j are expanded. For subsequently found equal-cost solutions, the maximum j may be updated to reflect the new upper bound, in case it has fewer decision variables. This procedure incrementally reduces the depth of j and consequently the number of paths that may be expanded. As soon as no viable nodes can be processed, because all are more expensive or equal-cost and at deeper levels, the search can be terminated. This strategy ensures that always the same m solutions are returned, because all paths are expanded that contain the m optimal solutions with the smallest number of decision literals.

5.7.2 Advanced Tree-depth Depending Termination

The previous strategy can further be improved by not only utilizing the tree-depth (vertical), but also the position within one level of the search tree (horizontal). For instance, all z optimal solutions share the same tree-depth, but only at most r alternative configurations are requested. Using the presented *tree-depth depending termination*, all z solutions have to be computed, since the sequence of finding them is unstable using multiple threads. The larger z is, the higher is the search overhead (in worst case $z - r$ avoidable nodes are expanded). To circumvent this, the multiple-criteria order of solutions described above is adapted by adding the number of left edges in the path from root to the node n. The number of left edges in a path is abbreviated with "left-branches" in the following. After having found r solutions, the minimum tree-depth as well as the maximum number of left-branches on that level are shared with all threads to prune paths. The value can be updated after having found another solution on a smaller or equal level. In the former case, the value is always updated because the new solution has a smaller tree-depth. In the latter case, the value is only updated if the number of left-branches is larger than the minimum of all currently hold solutions on that level. This strategy reduces the number of expanded nodes, when several solutions are on the same level.

5.7.3 Node Expansion Termination

For the presented cube-and-conquer approach (5.4), a simpler deterministic search termination detection can be used. The distribution of cubes after the first phase consistently assigns a set of work to each thread. Each worker thread processes the cubes in the same order without exchanging them, hence the initial search space partitioning is consistent for repeating configuration steps. This property can be exploited to terminate cubes early, by limiting the amount of nodes each thread is processing after r solutions have been found. For this, the sorting criteria is changed. Having two equal-cost solutions, the one that originated from an earlier cube is preferred.

6 Experimental Evaluation

This Section presents an evaluation and comparison of the developed algorithms in Chapter 5. As a baseline, the commercial product configurator *CAS Merlin* [6] is used. The evaluated algorithms are:

1. *Centralized Parallel A* Search* (CA*): Extension of the sequential A* search using multiple threads on a single OPEN list.
2. *Decentralized Parallel A* Search* (DA*): Extending the A* search by assigning an OPEN list to each thread. Nodes are exchanged among the workers.
3. *Parallel Cube-and-Conquer* (C&C): Two-phase approach. Firstly, the problem is decomposed into subproblems. Afterwards, the cubes are processed in parallel.
4. *Parallel Portfolio* (PP): Combining multiple sequential base solvers with different configurations.
5. *Parallel Portfolio A** (PPA*): Extending parallel A* search by using different settings for each worker thread.

All the algorithms are implemented in Java 11 and executed on the application server WildFly 15. The server has an Intel Core i7-7820HQ CPU and 16 GB RAM. We did experiments with up to 4 threads per SAT solve. This is not an impressive amount when compared to recent work in the area of parallel SAT solvers, nevertheless, we identified 4 as the maximum amount of threads that can be allocated to solving one configuration click (calculation of the next valid configuration) in order to use the available server capacities reasonably and economically in a commercial setting in our case.

We do not compare our algorithms to any existing academic implementations for similar problems in this paper for a few reasons. To facilitate a meaningful evaluation we would need to integrate state-of-the-art academic parallel PBO [42] or PMaxSAT [37] solvers into the configurator. This is difficult, since these solvers are written in C/C++ while our application is in Java. According to our preliminary evaluations, the translation and execution overhead is too big, especially for easy problem instances, which constitute the majority in our application. Additionally, our configuration problems also contain numeric constraints (like in SMT) which cannot be handled by PBO/PMaxSat. We tried integrating a MaxSMT solver but we could not obtain satisfactory results that way either. Not even for the sequential computation, let alone in parallel. Another reasons are the special requirements in our application such as deterministic calculation and enumeration of multiple optimal solutions.

Table 1 Comparison of different rule sets (Boolean formulae). RS1 to RS3 show formulae of industry cases. R3SAT are random 3-SAT instances and RSAT stands for random SAT instances. Ratio describes the clause to variable ratio of the resulting Boolean formulas.

Rule Set	Domain Complexity	# Clauses	# Variables	Ratio	Avg. clause length
RS1	High	17157	8483	2.02	3.98
RS2	Medium	4803	8318	0.58	3.34
RS3	Low	8023	1722	4.66	4.70
R3SAT	–	449	100	4.49	3
RSAT	–	850	100	8.50	5.8

The data used for the experiments consists of various real industry cases as well as random SAT and random 3-SAT instances[2]. The industry cases are divided into three groups (RS1, RS2, RS3) of various complexity, where RS1 is the most complex and RS3 the least complex[3]. The properties of the benchmarks are given in Table 1. The formula sizes are not as high as what we usually see in say SAT competition benchmarks. On the other hand, one must consider, that the time limit at the SAT competition is 5000 seconds while we want solutions in milliseconds. This is also related to using only 4 threads. Using many threads increases the overhead, which we cannot afford in our setting.

The entire system is tested using automated simulation of configuration processes. Each test for a rule set consists of n configuration steps. Every configuration step is repeated three times to have more accurate mean wall clock times. Therefore, for each rule set $3n$ data points are available. The usage of simulated configuration steps yields a mixture of small and larger configuration changes with varying response times. Considering randomly generated tests, 23 random SAT and 57 random 3-SAT instances are used (hence 80 data points). For each tested algorithm configuration, a warm-up phase is executed for the Java Virtual Machine (JVM). Despite measuring different metrics, the usage of automated tests also ensures the correctness of all implemented algorithms.

Our main goal was to conceptualize approaches that scale well with increasing numbers of processing units. In Table 2 the speedup of the different parallel algorithms is displayed using varying numbers of threads. We report relative speedup rather than absolute run-times, since we believe it is more representative and the standard way to evaluate parallel algorithms.

Largest improvements are attained on RS1 with an overall speedup of 2.5 for CA*. On the easier problem domains RS2 and RS3, the improvements decrease to 2.06 and 1.13. Especially the latter is explained by the numerous short running configuration tasks in RS3 that lead to overhead using CA* search. Results of DA* search show very similar effects, scaling well on complex rule sets but suffering from overhead on simpler ones like RS3. On RS3, the search algorithms perform better with two threads (0.91) than with four (0.90) due to the added synchronization and communication overhead. C&C's results do not show a clear pattern, it scales best on RS1 and RS3, less so on the medium rule set RS2. However, the addition of more threads shows larger diminishing returns. On RS1 the difference between three and four threads is only a speedup of 0.06 compared to the baseline. Furthermore, the table shows that the portfolio solver scales negatively in many cases with the number of processing units. An exception is the less complex rule set RS3. Hence, the scaling issues of the PP solver is related to the complexity of the problem instance.

[2] The JNH and CBS benchmarks from SATLIB `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

[3] The complexity of the groups RS1, RS2, and RS3 is defined based on the average runtime performance of the baseline (single threaded) algorithm on these benchmark sets.

Table 2 Comparison of overall speedup for parallel algorithms using varying numbers of threads. PPA* was evaluated only with 4 threads.

Rule Set		CA*	DA*	C&C	PP	PPA*
RS1	2 Threads	**1.63**	1.58	1.22	0.91	–
	3 Threads	**2.07**	1.98	1.37	0.88	–
	4 Threads	**2.50**	2.08	1.44	0.83	2.40
RS2	2 Threads	**1.58**	1.40	0.94	0.86	–
	3 Threads	**2.03**	1.43	1.01	0.72	–
	4 Threads	**2.06**	1.55	1.05	0.63	1.94
RS3	2 Threads	0.89	0.91	**1.12**	1.07	–
	3 Threads	1.01	0.90	**1.27**	1.15	–
	4 Threads	1.13	0.90	**1.33**	1.24	1.00
R3SAT/RSAT	2 Threads	1.40	1.49	**1.91**	1.02	–
	3 Threads	1.55	1.86	**2.26**	0.99	–
	4 Threads	1.72	2.06	**2.46**	0.97	1.72

All before presented results include the deterministic versions of the algorithms. To determine the approximate impact of the stricter search termination criteria, the best performing algorithm is considered: the centralized parallel A* search. For the evaluation, the most complex benchmark RS1 is used which contains a wide variety of configuration tasks. The impact is measured by the overall speedup of the non-deterministic version over the deterministic version of CA* search. We measured an overhead of approximately 2-3% on RS1. The overall performance cost is 2.4%, if only long running tasks (>100 ms) are considered.

A general observation is the increasing performance improvement on more complex rule sets. A main reason for this is the share of short running configuration tasks on less complex formulas (e.g. RS3). Having instances that can be solved very fast (in less than 50 ms), it is difficult to achieve significant improvements through additional processing units. The added overhead due to initialization and synchronization outweighs the benefits of solving the instance in parallel. Regarding the parallel portfolio, the solver performs worst on many industry cases. Moreover, it partly scales negatively with increasing thread pools. For each sequential base solver, the Boolean formula is duplicated. Performing k separate A* searches, with k being the number of threads, escalates the memory consumption due to the search tree construction. In most tests, the increased load on the system outweighs the benefits of having separate solver configurations that reduce the sensitivity to parameter tuning. Thus, the alternative portfolio approach PPA*, which applies different sets of parameters to parallel A* search, achieves more consistent and better results.

Further data (plots and tables) and discussion about the experimental evaluation are available in the main author's master's thesis [48].

7 Conclusion

Our goal was to find search algorithms that can exploit the capabilities of common multi-core processors while maintaining their completeness and determinism with respect to found solutions. To define the problem occurring in interactive configuration, the MinCostConf problem was introduced which extends the SAT and MinCostSAT problems and belongs

to the class of NP-hard problems. It describes the task of finding minimal-cost solutions given a start-configuration and user wish. Additionally, different custom cost functions were shown to model distinctive behaviors that evaluate configuration changes with respect to the start-configuration.

We presented various parallel algorithms to solve the MinCostConf problem. As a baseline, the existing sequential A* search algorithm was introduced with a custom cost function that prefers to keep prior user selections. Three major strategies for parallel search were implemented. Firstly, two versions of parallel A* search were conceptualized. Secondly, a parallel cube-and-conquer algorithm was presented and lastly, a parallel portfolio approach was proposed. To avoid search space duplication, an alternative was shown which applies portfolio concepts to parallel A* search. The introduction of parallel algorithms for MinCostConf added nondeterminism with regard to the order of found solutions. This has been addressed by designing robust search termination detection strategies which ensure that the search is only terminated when the expected configurations are found.

We performed experiments to compare the implemented algorithms using real industry cases as well as random SAT instances. The achieved speedup varied depending on the rule set, but for critical configuration tasks with longer response times, a consistent speedup between 2 and 3 was attained utilizing 4 worker threads. Lastly, the experiments also showed that deterministic behavior can be achieved with a reasonable overhead.

7.1 Future Work

There are several aspects in this paper that can be extended and further improved upon. Firstly, the evaluation was performed on a limited selection of rule sets using up to four threads. Thus, the presented algorithms can be optimized to utilize a larger number of processing units, although diminishing returns are expected.

Secondly, the presented algorithms are only a subset of possible approaches. Other algorithms that have been used in the literature can be adopted and changed to fit the presented problem. For instance, to limit the memory footprint the iterative deepening A* (IDA*) algorithm can be adapted ([32]). This can also improve the presented parallel portfolio approach which is limited by its resource consumption. Furthermore, the presented MinCostConf problem defines solutions as minimal-cost configurations. In some domains with very complex configuration models, this criteria may be loosened and only good but suboptimal solutions are requested. This could be performed for example with a parallel and deterministic version of beam search.

A reviewer of this paper suggested, that the methods used for robust and super solutions [21, 20, 15] in constraint programming bear a lot of similarity to our methods. In future work we would like to examine these similarities.

Lastly, parallel SAT related algorithms may also be used in other areas of interactive product configuration. Besides a valid configuration, additional information can be calculated, for example the attributes that are not possible to select without changing the pinned attributes. Therefore, these attributes may be grayed out for the user. To accelerate this calculation, a parallel algorithm can be applied. Another area of interest is multi-product configuration. Given several loosely coupled products, a user wish in one product can cause changes in other dependent ones, possibly causing a chain reaction. The calculation of this impact can be performed in parallel, by analyzing the impact of the user with the help of a dependency graph and performing independent sub-configurations in parallel.

References

1 Henrik Reif Andersen, Tarik Hadzic, and David Pisinger. Interactive cost configuration over decision diagrams. *Journal of Artificial Intelligence Research*, 37:99–139, 2010.

2 Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172, 2015.

3 Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20, 2005.

4 Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT competition*, 2013:1, 2013.

5 Csaba Biro, Gergely Kovasznai, Armin Biere, Gábor Kusper, and Gábor Geda. Cube-and-conquer approach for sat solving on grids. In *Annales Mathematicae et Informaticae*, pages 9–21, 2013.

6 CAS Software AG. Cas configurator merlin, 2020. URL: `https://www.cas.de/en/products/configurator/cas-configurator-merlin.html`.

7 Wahid Chrabakh and Richard Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37, 2003.

8 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

9 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

10 Matthew Evett, James Hendler, Ambuj Mahanti, and Dana Nau. Pra*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25(2):133–143, 1995.

11 Andreas Falkner, Alexander Felfernig, and Albert Haag. Recommendation technologies for configurable products. *Ai Magazine*, 32(3):99–108, 2011.

12 Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen. *Knowledge-based configuration: From research to business cases*. Newnes, 2014.

13 Eugene C. Freuder, Chavalit Likitvivatanavong, and Richard J. Wallace. Explanation and implication for configuration problems. In *IJCAI 2001 workshop on configuration*, pages 31–37, 2001.

14 Zhaohui Fu and Sharad Malik. Solving the minimum-cost satisfiability problem using sat based branch-and-bound search. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 852–859, 2006.

15 Begum Genc, Mohamed Siala, Barry O'Sullivan, and Gilles Simonin. Finding robust solutions to stable marriage. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 631–637. ijcai.org, 2017. `doi:10.24963/ijcai.2017/88`.

16 Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Møller, and Henrik Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. *small*, 10(1):3, 2004.

17 Youssef Hamadi, Said Jabbour, Cédric Piette, and Lakhdar Sais. Deterministic parallel dpll. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.

18 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: A parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010.

19 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

20 Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Robust solutions for constraint satisfaction and optimization. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 186–190. IOS Press, 2004.

21 Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Super solutions in constraint programming. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2004. doi:10.1007/978-3-540-24664-0_11.

22 Marijn Heule and Hans van Maaren. Look-ahead based sat solvers. *Handbook of satisfiability*, 185:155–184, 2009.

23 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65, 2011.

24 KEKIB IRANI and YI-FON SHIH. Parallel a* and ao* algorithms: An optimality criterion and performance evaluation. In *1986 International Conference on Parallel Processing, University Park, PA*, pages 274–277, 1986.

25 Mikolas Janota. Do sat solvers make good configurators? In *SPLC (2)*, pages 191–195, 2008.

26 Mikoláš Janota. *SAT solving in interactive configuration*. PhD thesis, Citeseer, 2010.

27 Mikoláš Janota, Goetz Botterweck, Radu Grigore, and Joao Marques-Silva. How to complete an interactive configuration process? In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 528–539, 2010.

28 Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

29 Yuu Jinnai and Alex Fukunaga. Abstract zobrist hashing: An efficient work distribution method for parallel best-first search. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

30 Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of sat solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.

31 Yoshikazu Kobayashi, Akihiro Kishimoto, and Osamu Watanabe. Evaluations of hash distributed a* in optimal sequence alignment. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

32 Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

33 Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *AAAI*, volume 88, pages 122–127, 1988.

34 Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *Asia and South Pacific Design Automation Conference, 2007*, pages 926–931, Piscataway, NJ, 2007. IEEE Operations Center. doi:10.1109/ASPDAC.2007.358108.

35 Chu Min Li and Felip Manya. Maxsat, hard and soft constraints. *Handbook of satisfiability*, 185:613–631, 2009.

36 Xiao Yu Li. *Optimization algorithms for the minimum-cost satisfiability problem*. PhD thesis, North Carolina State University, 2004. URL: https://repository.lib.ncsu.edu/handle/1840.16/4594.

37 Inês Lynce, Vasco M. Manquinho, and Ruben Martins. Parallel maximum satisfiability. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 61–99. Springer, 2018. doi:10.1007/978-3-319-63516-3_3.

38 Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

39 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

40 Hidetomo Nabeshima and Katsumi Inoue. Reproducible efficient parallel sat solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–138. Springer, 2020.

41 Mike Phillips, Maxim Likhachev, and Sven Koenig. Pa* se: Parallel a* for slow expansions. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.

42 Ted K. Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear optimization. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 283–336. Springer, 2018. doi:10.1007/978-3-319-63516-3_8.

43 Daniel Sabin and Rainer Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems and their applications*, 13(4):42–49, 1998.

44 Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: Parallel sat solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2010.

45 Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. Pasat – parallel sat-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001.

46 Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Detection of inconsistencies in complex product configuration data using extended propositional sat-checking. In *FLAIRS conference*, pages 645–649, 2001.

47 Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Ai Edam*, 17(1):75–97, 2003.

48 Nils Merlin Ullmann. Parallel sat-solving for product configuration. Master's thesis, KIT, Karslruhe, Germany, 2021. doi:10.5281/zenodo.5154040.

49 Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4-6):543–560, 1996.