Karlsruhe Institute of Technology

Cube&Conquer-inspired Malleable Distributed SAT Solving

Master's Thesis of

Maximilian Schick

at the Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

Reviewer:	Prof. Dr. rer. nat. Peter Sanders
Advisor:	M.Sc. Dominik Schreiber
Second advisor:	Dr. rer. nat. Markus Iser

01. November 2020 – 30. April 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 30.04.2021

. . . .

(Maximilian Schick)

Abstract

In order to make resource-efficient use of high-performance computing environments in the context of propositional satisfiability (SAT) solving, recent work suggested to resolve multiple formulae at once within a decentralized framework for job scheduling. In order to address the issue of highly variable processing times that are unknown in advance, malleable SAT solving methods have emerged. Malleability denotes the ability of a computational task to vary its number of processing resources during execution. In this work we extend the field of malleable SAT solving by introducing two new methods that are inspired by Cube&Conquer. This paradigm is based on using special lookahead heuristics in order to partition a SAT instance into sub-problems that can be solved simultaneously. Our first method uses malleability to resource-efficiently create a static workload which is afterwards balanced across a flexible number of processing elements. The second method is built around continuous work generation coupled with work stealing in order to exploit a fluctuating amount of resources. To enable collaborative SAT solving of a dynamic workload, we present a new decentralized approach based on exchanging failed assumptions. In the evaluation, we inspect parameters and show that both methods can outperform a state-of-the-art sequential SAT solver. We also compare them to an existing malleable solving engine based on portfolios. All work was implemented within the recently published *Mallob* framework that manages the scheduling and load balancing of a fluctuating number of malleable tasks within a distributed computing environment.

Zusammenfassung

Zur ressourceneffizienten Nutzung von Hochleistungsrechnern für die Lösung des Erfüllbarkeitsproblems der Aussagenlogik (SAT) wird in neueren Arbeiten vorgeschlagen, den Ablauf der Bearbeitung mehrerer Formeln von einem dezentralen System verwalten zu lassen. Um die Problemeatik der hochvariablen und im Voraus unbekannten Verarbeitungszeiten zu behandeln, haben sich *malleable*-SAT-Löseverfahren bewährt. Malleability bezeichnet die Fähigkeit einer Berechnung, während ihrer Ausführung die Anzahl der zugewiesenen Rechenressourcen zu verändern. Diese Arbeit erweitert den Bereich der malleable-SAT-Löseverfahren, durch die Einführung zweier neuer Methoden, die von Cube&Conquer inspiriert sind. Dieses Paradigma basiert auf der Partitionierung einer SAT-Instanz mittels vorausschauender Heuristiken. Die resultierenden Teilprobleme können dann simultan gelöst werden. Die erste Methode verwendet Malleability, um ressourceneffizient eine statische Arbeitslast zu erzeugen, die anschließend auf eine dynamische Anzahl von Verarbeitungselementen verteilt wird. Die zweite Methode basiert auf einer kontinuierlichen Arbeitsgenerierung in Kombination mit Work-Stealing, zur Ausnutzug einer variabel großen Rechenkapazität. Um kollaboratives SAT-Lösen in Verbindung mit einer dynamischen Arbeitslast zu ermöglichen, wird ein neuer dezentralen Ansatz vorgestellt, der auf dem Austausch von fehlgeschlagenen Annahmen basiert. Die Evaluierung betrachtet mögliche Parameter und zeigt, dass beide Methoden einen hochmodernen sequentiellen SAT-Löser übertreffen können. Zusätzlich werden beide mit einem existierenden malleable-SAT-Löseverfahren verglichen, das Portfolios verwendet. Die gesamte Arbeit geschieht im Rahmen des kürzlich veröffentlichten *Mallob*-Frameworks. Dieses verwaltet den Ablauf der Ausführung einer schwankenden Anzahl von malleable-Berechnungen auf verteilte Verarbeitungselemente.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The *Boolean satisfiability problem*, or SAT problem, is one of the most prominent problems in theoretical computer science. It is defined as the question of whether there is a variable assignment for a propositional formula such that it evaluates to true. If such an assignment is found, the associated formula is regarded as satisfiable. The nonexistence of a satisfying assignment makes a formula unsatisfiable. The SAT problem was the first one to be proven NP-complete, thus defining an entirely new complexity class [9]. Being the original NP-complete problem, places it in the center of the families of NP and NP-hard problems. An efficient solving algorithm would therefore find application in a multitude of fields. This has drawn a lot of attention towards possible solving methods [8]. Due to the NP-complete nature of the problem, all solving algorithms require worst-case exponential time [15, 23]. However, there has been promising work on creating solvers that work well in practice. These have been applied successfully to SAT instances from fields like integrated circuit design automation, model-checking and planning [8, 23].

The interest in SAT solving culminated in the founding of the International SAT Competition in 1992. The contest, which is held annually since 2002, allows researchers to compare their approaches with other new ideas and state of the art techniques. This has contributed significantly to the rapid progress in the field of SAT solving, making solvers more robust, more reliable and more efficient to solve a wide variety of problem instances [20, 27].

The advances in SAT solving over the last decades have resulted in very effective state-of-the-art solvers. They are capable of solving many problems arising from industrial applications in a reasonable amount of time. This has been achieved by incorporating advanced heuristics and numerous preprocessing and inprocessing techniques. In addition, specialized data structures have been used to push the maximum performance on today's single-core architectures [3]. To take advantage of this progress, a lot of research has been done to transfer other problems from scientific fields such as graph theory, computer science, computer engineering, and operations research to equivalent SAT instances [7]. The resulting positive loop has made the SAT problem even more relevant [8].

However, the widespread application of SAT solvers on numerous tasks had an impact on the average problem instance. Modern solvers are challenged by problems that are larger, more complex and more diverse in their structure. This development has limited the general increase in their performance [16, 1]. To cope with this issue and to utilise the prevalence of multi core systems, much work has been done recently on parallel SAT solving solutions. The goal of this next step is to create a new leap in SAT solver performance and to find ways to solve industrial problems that are currently not solvable in a reasonable amount of time. The Parallel Track of the SAT Competition has therefore been an integral part since its inception in 2008 [16]. Parallel SAT solving is usually done by combining instances of one or multiple single threaded state-of-the-art solvers. This allows the creation of a parallel SAT solver without having to worry about the complicated internals of a modern sequential SAT solver. The solver can simply be used in a black box manner utilizing the expertise gained from decades of research.

There are two predominant approaches for parallel SAT solving: portfolios and search-space splitting approaches [24]. Most relevant parallel solvers can be assigned to one of these two classes or use a hybrid solution.

- Portfolios exploit concurrency by applying several complementary strategies to a problem. This is done by using a single or multiple solvers in a variety of configurations. The approach is based on the idea that one of the strategies used might be particularly well suited to solve the problem. The problem itself is not modified.
- Search-space splitting is based on the concept of splitting the given SAT problem into several smaller disjoint problems and solving them in parallel. Splitting can be done either in advance, thus creating an additional upfront task or dynamically during solving, making the algorithm much more complex. Finding a good split of a problem is a complex task in itself. It is difficult to find a split which produces sub-problems of reduced difficulty and does not merely multiply the work to be done.

Both approaches can be improved by having concurrent solvers share learned information which simplifies the formula.

Recently, massively parallel and distributed SAT solving gained more attention. This has lead to the introduction of a Cloud Track in the SAT Competition 2020 [4]. Having access to a cluster consisting of hundreds of multi core machines could give state-of-the-art parallel SAT solvers the edge to solve even more complex problem instances. It could also enable the creation of a resource-efficient SAT solving cloud service for a fluctuating number of problem instances by scaling each concurrent solving process based on its demand and the formula's priority. This is possible by introducing *malleability* to parallel SAT solving [25]. Malleability in a parallel context stands for the ability to dynamically alter the amount of processing elements in an application during its execution [12]. Instead of setting a fixed degree of parallelization at startup, it can be increased or decreased during the run time of a malleable computing task. This idea was already brought to the portfolios approach in a submission to the 2020 SAT Competition [25], albeit only applied to a single formula at a time due to the rules of the competition.

1.2 Approach

This thesis focuses on our progress in introducing malleability to parallel SAT solving based on search-space splitting. To our knowledge, this has only been done in Paracooba [17], but without priority-based load balancing and only for static workloads. Our approaches incorporate priority-based load balancing and our second approach is built around managing a dynamic workload. We do this in the scope of Mallob [26], a decentralized platform that handles the scheduling and load balancing of a fluctuating number of malleable tasks within a distributed computing environment depending on the task's priority and demand. All presented methods are therefore built around Mallob's specialized communication structure of the processing resources that are assigned to a task. A task in this context represents the solving of a single instance of the SAT problem. Another limitation is our design decision to base our approaches around a solver that implements the IPASIR interface [5]. This allows the resulting methods to be used with various state-of-the-art solvers. It also contributes to the simplification, because we will not engage in the internals of the SAT solver. We propose the following two different SAT solving methods:

- The first one uses an initial sequential step to split the given problem into a predefined number of sub-problems. In the second step, these are distributed to concurrent SAT solvers in the assigned processing resources. The balancing of the workload is then based on the fact that there are more sub-problems than active solvers. This compensates for different processing times per sub-problem.
- The second method is more dynamic in nature. Its workload balancing is based on enabling each processing element to split a problem into sub-problems. This allows each of them to divide received sub-problems further, providing diverse work for the local solvers. Therefore, it is guaranteed that each inactive solver can obtain an unsolved and unique sub-problem. A given problem is so dynamically partitioned to the extent necessary to provide work to each solver.

Both methods are tested in different configurations to estimate the influence of used parameters and to test how well their solving capabilities scales with an increasing number of processing elements per task. Then we compare the solving performance of both with the the portfolio-based solving engine included in Mallob [26].

1.3 Structure of this thesis

We begin in Chapter 2 with the definition of the SAT problem and give a brief overview of modern sequential SAT solving techniques. Next, we explain the search-space splitting approach to parallel SAT solving and its caveats and propose possible implementations. We end the Chapter with a reference to two related works. In Chapter 3, we introduce Mallob and define the interface a task must implement to allow its malleable scheduling. In Chapter 4, we state and reason for all the chosen constraints on our approaches. Our two approaches are presented in the Chapters 5 and 6 and we evaluate them in Chapter 7. Finally, we summarize our results in Chapter 8 and propose future work.

2 Preliminaries and Related Work

In this chapter we first give a more detailed definition of the SAT problem and reference the associated terminology that is used in the context of SAT solving. An explanation of the basic algorithms and their properties follows. After that, we give an overview of the search-space splitting approach to parallel SAT solving, list its weaknesses and present two possible implementations. Finally, we introduce two related works that apply this approach in a distributed computing environment.

2.1 Definitions

We start by giving some basic definitions that will help us to specify parts of the Boolean satisfiability problem.

Definition 1 (Boolean variable)**.** A *Boolean variable* or *propositional variable* or in further mostly just *variable* is a placeholder for the truth values *true* and *false*. In this thesis we use them in combination with the unary operation *negation* ($\neg$) and the binary operations *disjunction* ($\vee$) and *conjunction* ($\wedge$).

Definition 2 (Boolean formula)**.** A *Boolean formula* is a function mapping each assignment of a finite set of Boolean variables to either true or false.

Definition 3 (Literal)**.** A *literal* represents a positive assignment of a variable, a negated variable is represented by an *inverse literal*. We use literals to form Boolean formulas and to symbolize variable assignments.

Definition 4 (Clause)**.** A *clause* is a disjunction of one or multiple literals and inverse literals.

Definition 5 (Conjunctive normal form (CNF))**.** A Boolean formula is in the *conjunctive normal form* if it consists of a single conjunction of one or multiple clauses.

Definition 6 (Unit propagation)**.** *Unit propagation* means affecting the entire formula by a chosen variable assignment x. This is done by manipulating the clauses. All clauses that contain x can be removed. They are satisfied by this partial assignment. All clauses that contain $\neg x$ remove the literal. This procedure allows to simplify a formula as assignments are made. It also helps to find dependant variable assignments that are forced by previous assignments.

2.2 The Boolean satisfiability problem

The Boolean satisfiability (SAT) problem is defined as the question of the existence of a satisfying variable assignment for a given Boolean formula. In the context of SAT solving, all formulas are required to be in conjunctive normal form. In this structure, it is trivial to see that a satisfying assignment has to satisfy each individual clause. The clauses therefore play an essential role in SAT solving. Each clause consists of a subset of the finite set of variables of the formula. Each variable can be contained exactly once per clause. We give an example of a Boolean formula in CNF in combination with a variable assignment below. The assignment proves the satisfiability of the formula. It can be verified by simply checking if each clause is satisfied.

$$\text{Formula} = \underbrace{(\overset{\substack{\text{Literal}\\ \downarrow}}{x_1} \lor x_2 \lor \overset{\substack{\text{Inverse literal}\\ \downarrow}}{\neg x_3})}_{\text{Clause}} \land (\neg x_1 \lor \neg x_2) \land (x_1) \land (x_1 \lor x_3)$$

$$\text{SAT assignment} = \{x_1, \neg x_2, \neg x_3\}$$

2.3 SAT solving algorithms

In this section, we present sequential and parallel SAT solving algorithms and methods and their properties.

2.3.1 SAT solver completeness

Solvers for the Boolean satisfiability problem are called SAT solvers. They can be divided into two categories: *complete* and *incomplete algorithms*. A complete algorithm is designed to eventually exhaust the entire search space of a given problem. Thus, it is guaranteed that the satisfiability will ultimately be determined. However, the exhaustion of the search-space is only possible in exponential time. An incomplete algorithm does not offer this guarantee. In this work, we exclusively focus on solvers that use complete algorithms and combine them to create new complete algorithms.

2.3.2 Davis–Putnam–Logemann–Loveland algorithm

The first SAT solving algorithm was proposed by Davis and Putnam in 1960 [11] and improved by Davis, Logemann and Loveland in 1962 [10]. It is therefore known as the *Davis–Putnam–Logemann–Loveland (DPLL) algorithm*. It is based on doing a depth-first search by heuristically choosing a Boolean variable of the formula and assigning a truth value to it, thereby branching into a sub-search-space. This is done by propagating the decision. If propagation has resulted in an unsatisfiable formula, the algorithm backtracks to the nearest non-conflicting assignment branch. Further, if each branch has resulted in a conflict, the formula is considered unsatisfiable. Otherwise, a satisfying assignment is found eventually.

2.3.3 Conflict driven clause learning

The majority of modern sequential SAT solvers are based on the *conflict-driven clause learning* (CDCL) algorithm. The method is complete and has proven to be very efficient in solving large but simple instances of the SAT problem, making it especially useful in practical applications. CDCL is an expansion of the Davis–Putnam–Logemann–Loveland algorithm that was used in earlier SAT solvers. The core invariant of DPLL is kept and expanded with the incorporation of a more complex conflict analysis step. This step allows the extraction of the information that lead to a contradiction. The information is then used to simplify the formula by adding redundant clauses and to enable non-chronological backtracking. A detailed description of the functionality of a CDCL based SAT solver can be found in [7].

2.3.4 Incremental SAT solving

In many applications, SAT solvers face the challenge of solving incrementally expanding problems or a single problem under several different partial assignments. To accomplish this, a fresh solver instance is applied each time. This results in the full duration per run. In order to perform this task more efficiently, solvers should be able to apply learnt simplifications on related problem instances. This would allow the reuse of the same solver instance for all problem instances. The simplifications that were learned while solving the first formula should then be usable to reduce the run time from the second instance onwards. This feature is known as incremental SAT solving. The applicability in multiple domains has lead to most state-of-the-art solvers including it in some form.

To compare the progress and to standardize the interface of incremental SAT solving, a dedicated Incremental Track was added to the SAT Competition in 2015 [5]. The proposed interface to be implemented is called IPASIR which is the reversed acronym for "Re-entrant Incremental Satisfiability Application Program Interface". In the Table 2.1 we present methods from the interface. This is necessary because both approaches in this thesis are built around this interface and thus reference its methods in their descriptions.

2.3.5 Lookahead solver

A *lookahead* solver is also a complete SAT solver that is based on the DPLL-algorithm. But instead of focusing on a better conflict analysis like in CDCL, they use a sophisticated lookahead procedure to determine better branching variables. Given the formula F a lookahead on the possible variable x works as follows [18]: First the decision x is propagated on F creating the reduced formula F'. If there was no conflict, the reduction from F to F' is measured. There are several heuristics for this measurement. A widely used one is the proportional number of clauses that have changed because of the propagation to the total number of clauses. This step is then repeated with $\neg x$. If the propagation has lead to a conflict, the variable is remembered as *failed literal*. This procedure is performed for all promising branching variables. Finally, the algorithm branches on the variable that had the largest impact on the formula in both polarities. The sophisticated approach of lookahead

Method	Behavior
add_I	Permanently add an additional clause to the formula.
$assume_I$	Assume a partial assignment for the next call to $solve_I$.
$solve_I$	Start to solve the formula consisting of all clauses that were added, under the specified assumptions. If the formula is satisfiable, SAT is returned. If the formula is unsatisfiable, UNSAT is returned. Otherwise UNKNOWN is returned.
$failed_I$	If the formula was proven to be unsatisfiable under a given partial assignment in the last call to $solve_I$, this method can be used to check which assumed literals were used in the proof and therefore are responsible.
$terminate_I$	This method allows to insert a predicate into the solving loop which can be used to force an early termination.

Table 2.1: Overview of the IPASIR interface.

solvers has proven to be efficient in solving small but very complex problems [18]. In addition, the structure of the used algorithm is well suited for parallelization. However, for large industrial problems, a single-threaded CDCL solver outperforms a parallelized lookahead solver running on multiple cores [18].

2.3.6 Search-space splitting

Search-space splitting is one the most dominant approaches to parallelize CDCL solvers. It is based on partitioning the search space of the given SAT problem into several disjoint sub-spaces, each defining a sub-problem. All sub-problems can be processed simultaneously. This is done by selecting an internal variable of the formula and thereby defining the two sub-formulas where the selected variable has been assigned one of the two truth values. For example, if we have the formula F that contains the variable x_1 we can create the two disjoint sub-formulas F_1 and F_2 like shown below. The original formula F can be recreated by forming the disjunction of both sub-formulas.

$$\underbrace{(x_1 \land F)}_{F_1} \lor \underbrace{(\neg x_1 \land F)}_{F_2} = F$$

This division step can be repeated on the sub-formulas as often as desired. In this way, a binary tree can be created where each node represents a partial assignment and each edge depicts the assignment of a single variable. The leaves then form the set of the smallest disjoint sub-problems that are equivalent to F in a disjunction. This is depicted in the example below.

Example. Let F be a formula. We want to use search-space splitting on F to create four sub-formulas. We therefore have to apply the presented division step three times. This results in the creation of the binary tree below.

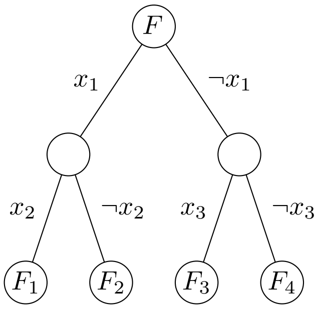


Figure 2.1: Illustration of the creation of the sub-problems.

The created sub-formulas are

$$\begin{aligned} F_1 &= (\ \ x_1 \wedge \ \ x_2) \wedge F, \\ F_2 &= (\ \ x_1 \wedge \neg x_2) \wedge F, \\ F_3 &= (\neg x_1 \wedge \ \ x_3) \wedge F, \\ \text{and } F_4 &= (\neg x_1 \wedge \neg x_3) \wedge F \\ \text{with } F &= F_1 \vee F_2 \vee F_3 \vee F_4 \end{aligned}$$

Multiple instances of a CDCL solver can then be used to concurrently work on each leaf. If a single instance finds a satisfying assignment for a sub-formula, the entire problem is solved. This follows from the disjunction that is needed to recreate the original formula. The satisfying assignment of the sub-formula can simply be expanded using the literals on the edges towards the root to create a valid satisfying assignment for the original problem. In order to prove unsatisfiability, all leaves need to be proven unsatisfiable.

However, finding optimal division variables is a hard problem itself [22]. Each division step must generate two sub-problems that are individually easier to solve than the outgoing problem. Both sub-problems should also be equally time consuming to solve. Otherwise, it would badly affect the load balancing.

The following two pathological cases show how poor partitioning can affect a formula and thus its solving using the search-space splitting approach [2]. Let F' be a formula in CNF. Using the two additional variables $x_1, x_2 \notin F'$, we can define another formula F as follows that is also in CNF.

$$F = ((x_1 \vee x_2) \wedge F') \wedge ((x_1 \vee \neg x_2) \wedge F')$$

1. If x_1 is chosen for a division, the sub-formulas

$$\begin{aligned} F_1 &= \ \ x_1 \wedge ((x_1 \vee x_2) \wedge F') \wedge ((x_1 \vee \neg x_2) \wedge F') = \ \ x_1 \wedge F' \\ \text{and } F_2 &= \neg x_1 \wedge ((x_1 \vee x_2) \wedge F') \wedge ((x_1 \vee \neg x_2) \wedge F') = \neg x_1 \wedge (x_2 \wedge F') \wedge (\neg x_2 \wedge F') \end{aligned}$$

are created. It is easy to see that F_2 can be proven to be unsatisfiable without having to traverse F', making the associated solving task very light. In a parallel context, this leads to inefficient load balancing.

2. If x_2 is chosen for a division, the sub-formulas

$$\begin{aligned} F_1 &= x_2 \wedge ((x_1 \vee x_2) \wedge F') \wedge ((x_1 \vee \neg x_2) \wedge F') = x_2 \wedge x_1 \wedge F' \\ \text{and } F_2 &= \neg x_2 \wedge ((x_1 \vee x_2) \wedge F') \wedge ((x_1 \vee \neg x_2) \wedge F') = \neg x_2 \wedge x_1 \wedge F' \end{aligned}$$

are created. This split forces the solvers that work on F_1 and F_2 to traverse the unmodified formula F'. The division has therefore resulted in two redundant tasks. This is bad because redundant task clog up processing resources with unnecessary work.

Because of such cases, the splitting variables should be carefully selected. In particular, when handling formulas that are likely to be unsatisfiable, since the algorithm requires each associated sub-formula to be proven unsatisfiable in order to determine UNSAT.

Choosing optimal division variables requires the use of heuristics [22]. In the literature, possible heuristics fall into the following two categories:

Lookahead heuristics Lookahead heuristics are based on predicting the behavior of a solver if a given variable would be assigned. This is done exactly as in the lookahead solver from Subsection 2.3.5. Each variable x_i is rated according to how an assignment of x_i or $\neg x_i$ would affect the formula. The variables that have a large impact on the formula in both polarities are then preferred for partitioning. The advantage of this heuristic is that it can be determined in advance. However, it demands a non-negligible amount of additional computing time.

Lookback heuristics Lookback heuristics are based on gathered statistics during a past behaviour of a sequential SAT solver. This is done by letting a solver instance work on a formula for a certain amount of time and then extracting variable-related information. A potential statistics is the number of times the assignment of a variable was inversed during branching. This type of heuristic can be used to find good partitioning variables during solving without requiring additional work.

2.3.7 Cube&Conquer

Cube&Conquer is a SAT solving method that can be seen as an implementation of the search-space splitting approach. The term was coined in the work [18] by Heule, Kullmann, Wieringa and Biere. Its main idea is to interleave a lookahead solver and an CDCL solver in a two phase solving algorithm. In the first phase, a state-of-the-art lookahead solver is used to partition the formula into sub-problems that are easy to solve for the CDCL solver. This works well because of the sophisticated heuristics that are used in a lookahead solver. This approach is therefore based on lookahead heuristics. To use a lookahead solver in this way, its internal algorithm has to be modified. Instead of depth-first following each decision branch until either a conflict or a satisfying assignment is found, each branch is cut off at a preset depth. A cut off branch is then ignored in the further branching and cannot be backtracked to. This can be compared to a branch that has been closed because of the associated sub-formula being proven unsatisfiable. If this modified algorithm is applied to an instance of the SAT problem, a perfect binary tree of the given depth is created. The leaves then define a disjoint set of sub-problems which are called the *cubes*. Since the goal is to create sub-problems that are easy to solve for a CDCL solver, combinatorial hard problems may need to be split into thousands or millions of cubes.

In the second phase, multiple concurrent instances of a state-of-the-art CDCL solver are used to solve each individual cube. This can be done especially efficient if they feature incremental SAT solving. The same solver instance may then be sequentially applied to multiple cubes by interpreting the given partial assignment using its assume functionality. This allows reusing simplifications that were found during the solving of a previous cube.

In practice, this approach has proven to be very effective for large and complex problems. A reason for this is that both solver types are able to make use of their individual strengths. The lookahead solver is capable of finding good divisions using its sophisticated heuristics. This works well in avoiding the pathological cases that are given in Subsection 2.3.6. The CDCL solver is then effective in solving the resulting large but now less complicated sub-problems.

2.3.8 Divide&Conquer with work stealing

Divide&Conquer is another implementation of the search-space splitting approach [24, 2]. Compared to Cube&Conquer, it can be considered more dynamic since it is not based on a static prepartitoned workload. The approach is based on multiple concurrent SAT solvers, each working on a sub-problem. If a solver s_1 becomes idle, work stealing is used to provide it with new work. This requires another solver s_2 to further divide its sub-problem using the division step from Subsection 2.3.6. The used splitting variable is usually chosen based on lookback heuristics. This allows s_2 to quickly determine a good variable on the basis of its solving progress. The prompted solver s_2 can then also use the chosen variable to simplify its sub-formula. The workload balancing of this approach is based on the assumption that eventually each solver is working on a sub-formula that requires a reasonable amount of work.

2.4 Related work

In this section we present work related to ours.

2.4.1 Paracooba

A related work is the state-of-the-art SAT solver Paracooba [17]. It uses the Cube&Conquer method and is designed for a distributed computing environment. The solver consists of master nodes that initiate solving and worker nodes that do actual solving using a local pool of CDCL solvers. All nodes are connected and communicate by sending messages over UDP and TCP. Additional nodes may be added dynamically using a custom auto discovery protocol. To solve a problem with Paracooba, the associated master node must first split it into cubes using the external lookahead solver March [19] or the integrated CDCL solver Cadical [13]. All compute nodes then parse both the formula and the cube tree. The distribution of work is then based on each node handling tasks representing paths in the cube tree. Nodes distinguish between assigned tasks (path to leaves) that a node is working on, and unassigned tasks (path to an inner node) that can be explored deeper to open up more paths, or distributed to another node. This process is started by the master node locally creating the unassigned task representing the empty path to the root of the cube tree. Each node is also responsible for returning the results of distributed tasks. This allows the master node to eventually determine whether the problem is satisfiable. The distributed paths of the cube tree are encoded as 64-bit unsigned integers to ensure minimal bandwidth requirements. Paracooba performs malleable load balancing by balancing the workload given by the tasks from multiple concurrent master nodes across all available worker nodes.

The basic principle of our first approach is similar to Paracooba. However, we use a centralized workload balancing strategy by creating a designated manager that directly distributes cubes to workers. We also allow for the same cube to be spread to multiple workers. This created redundancy helps in case a worker exits the solving process.

2.4.2 Ampharos

Another related work is the SAT solver Ampharos [1]. It is based on the Divide&Conquer method and is also designed for a distributed computing environment. However, it does not use the presented work stealing strategy. Instead, each solver works on a leaf of a binary tree of sub-formulas like in the example from Subsection 2.3.6 that is created in a distributed manner and managed by an designated manager. Each solver instance can work on any leaf. If a solver decides to work on a leaf, they do so until a predefined number of conflicts is reached. If this number is exceeded, the solver may choose to work on another leaf. Each leaf posses a counter for the number of failed solving attempts. If this counter exceeds a certain threshold, the cube is divided further. The used variable is chosen based on lookback heuristics from its last attempting solver. Each cube is therefore split until it can be solved in the given conflict limit. If a leaf is proven to be unsatisfiable, the associated branch is closed. This is then announced globally. A closed branch is ignored when searching for a new leaf to solve.

The method starts on a single leaf that represents the unmodified formula. It terminates if a leaf is proven to be satisfiable or if all leaves were closed. Additionally, the solvers share learnt clauses to improve their solving performance. Ampharos was originally created without malleability in mind. However, the distributed and independent nature of the used algorithm seems to be well suited for the inclusion of malleability.

We created our second approach with the dynamic cube generation of Ampharos in mind. However, we decided to use separate cube generators based on lookahead heuristics in order to not depend on regularly interrupting our solvers. The reason for this design decision is given in Chapter 4. Also, we did not use a designated manager to prevent a communication bottleneck in a massively parallel setting. Instead, we have developed invariants and specialized communication strategies that allow each processing element to control and extend a held set of cubes. To enable collaborative SAT solving, we rely on a sharing strategy of found failing partial assignments.

3 The Mallob Platform

In this chapter we introduce *Mallob*, the **Mal**leable **Lo**ad **B**alancer or **M**ulti-tasking **A**gi**l**e **Lo**gic **B**lackbox [26]. Mallob is important in the context of this work because we used it as a platform to execute our malleable algorithm in a distributed context. We will therefore explain how Mallob works and how it controls the malleability of its running jobs. We will then present our approaches using Mallob-specific terminology.

3.1 Basic concept

The scope of Mallob is a distributed computing environment consisting of m equivalent *compute nodes* that are able to exchange messages using a Message Passing Interface [14]. Each compute node is used to host $c \geq 1$ separate MPI processes. These $p := m \cdot c$ processes are called *processing elements* (PE). Each PE has a unique global rank r and encapsulates multiple cores. A PE embodies the smallest computational resource in the context of Mallob. Because of this we specify every demand of processing resources in a number of PEs. An illustration of the architecture is given in Fig. 3.1.

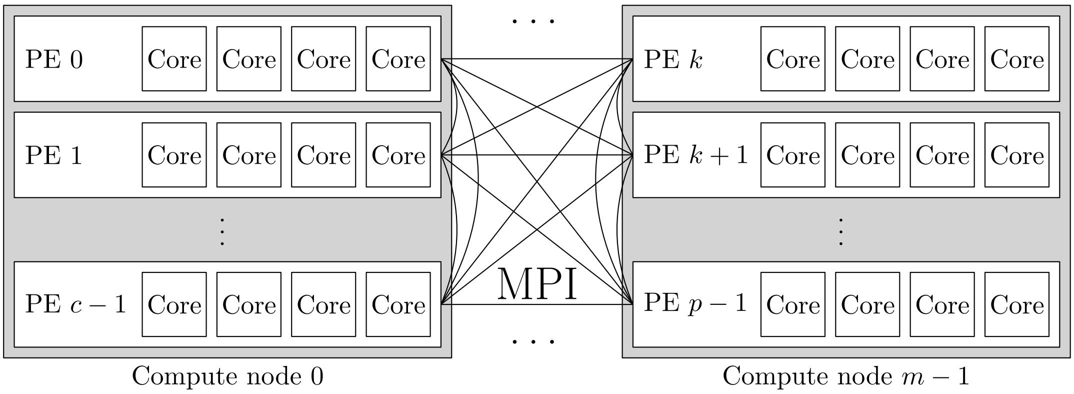


Figure 3.1: An overview of the distributed architecture of the Mallob platform [26].

The Mallob platform is designed to handle the massively parallel and distributed scheduling and processing of a fluctuating number of malleable jobs on p PEs. A *job* in this context is defined as a single task that provides procedures that allow it to be solved by several largely independent job nodes. These jobs may enter the platform through a given interface at any time. Mallob assumes that there are always more processing elements than active jobs. A *job node*, or node in short, can be viewed as a single processing element that is currently working on a job. Each job j is introduced to the system with a preset *priority* $\pi_j \in (0, 1]$

and may dynamically change its *demand* $d_j \in \mathbb{N}$ of assigned PEs. By default, the demand of each job starts at $d_j = 1$ and is doubled in intervals given by a system-wide growth parameter. This can be overwritten by setting a constant demand or by implementing a custom function of growth. The malleability of a job can then be used to dynamically vary the number of assigned job nodes according to its priority, its demand and the overall system state.

Internally, a job j is represented as a binary tree, called the *job tree* T_j, consisting of all associated job nodes $p_x(j)$. The root node $p_0(j)$ is the first node that was assigned to the job, marking the beginning of its life cycle. This node is responsible for the representation of the job j in the system and to inform the system of the job's resource requirements d_j. Consequently, it cannot be reassigned during the life cycle of a job. Apart from that, it behaves like all the other nodes. Each node $p_x(j)$ in the T_j has an unique index $x \geq 0$. The children of $p_x(j)$ have the indices $p_{2x+1}(j)$ (left child) and $p_{2x+2}(j)$ (right child). If a job node is assigned to a job, it is added to the tree at the lowest free index. Conversely, if a job needs to reduce its number of job nodes, the leaf with the highest index is removed first. Because of this, T_j is always a complete binary tree. Each node locally stores the global rank of the root, its parent and its children. These *references* allow each PE to address these other PEs that work on the same job via MPI messages. An illustration of how multiple PEs form a binary job tree is given in Fig. 3.2.

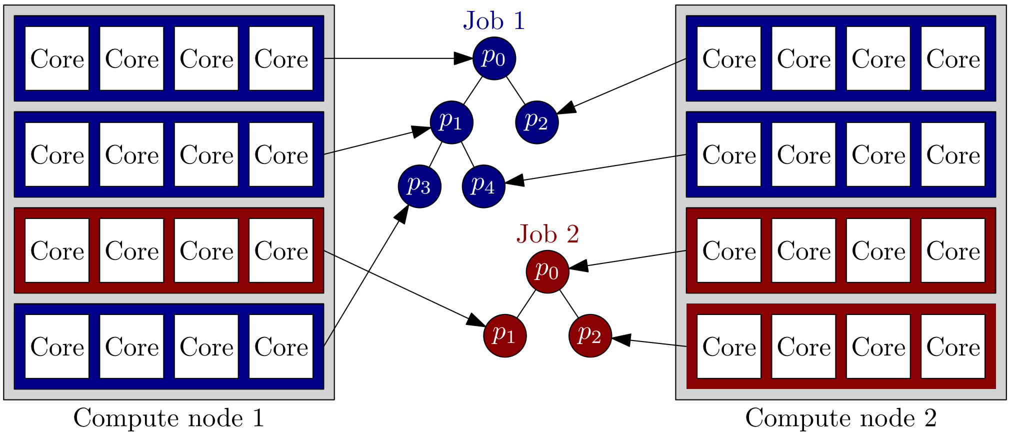


Figure 3.2: Schematic illustration of two job trees and their assigned PEs from two identical compute nodes.

The structure of a binary tree can be used for very efficient data sharing over all PEs of a job. If there is information across all nodes which we would like to concentrate to form a single piece of data, we can simply perform a step-wise aggregation from the leaves towards the root. If we then would like to share this data with all nodes, we can do so by broadcasting it from the root to the leaves. Both operations only require a linear communication effort per node depending on the size of the sent message. They are therefore suitable for a massively parallel setting. Both steps of this processes are illustrated in Fig 3.3.

The amount of nodes in a job tree T_j is given by the *volume* of its job $v_j \geq 1$. These volumes are calculated system-wide, taking into account all active jobs, in distributed

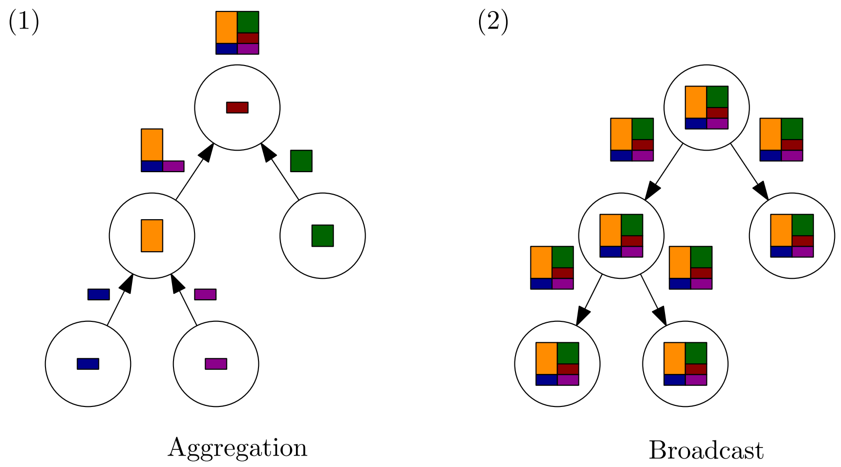


Figure 3.3: Exemplary data flow over all nodes of a job. Step one shows a step-wise aggregation, step two a broadcast of the aggregated piece of data.

load-balancing computations. If a job j was assigned the volume v_j, its job tree T_j consists of the nodes $p_0(j), \dots, p_{v_j-1}(j)$. If v_j changes, the new value is recursively broadcast in the tree starting from the root $p_0(j)$. If the volume was decreased and the tree therefore has to shrink in size, each excess node $p_x(j), x \geq v_j$ starts to leave the job on receiving the new volume. Leaving a job is accomplished by suspending all work on the job by following a job-specific suspension procedure. The broadcast allows the parents of nodes with $p_x(j), x \geq v_j$ to remove their child references accordingly. In the opposite case, when the tree is supposed to grow, the designated parent of the new nodes sends a request to currently inactive PEs to join the tree. These requests are preferably sent to former job nodes. This strategy allows the re-use of previously created solving resources and can thus reduce overhead and make use of the progress which a job node made before it was suspended. If a PE that was previously active on a job j accepts a participation request for j, it follows a job-specific resumption procedure. If the PE is new to the job, it has to perform a start procedure. After a node was added to the job tree, the references in its parent are updated. In practice, we allow each PE to keep the solving resources of a certain number of jobs and enforce it to discard the resources of old jobs if this limit is exceeded.

3.2 Designing a malleable job

To allow the malleable scheduling of a job to a varying number of PEs, a Mallob-specific job interface needs to be implemented. This interface contains the start, suspend and resume procedures that are required during job volume changes. It also contains various queries to allow the platform to check the status of a job on a PE. To understand the design of the approaches presented in this thesis, it is required to introduce the reader to the lifecycle of a job on a PE and how the PE interacts with it. In the following we will therefore go into detail and describe the interface of a job and all assumptions the system makes upon them. We will also go over the internal processes of a PE.

Each processing element r specified by its rank $0 \leq r < p$ hosts a single controlling thread t. A main function of t is to listen for and react to job requests. This allows it to control r accordingly. In addition, the thread t is the only running entity on r that has the authority to send and receive messages. Working on a job j requires t to first create a job context $\kappa_j(r)$. Such a context contains a local instance of the job's interface. The context $\kappa_j(r)$ is created if t has received a message to do so. The creation of a new job context in r does not interfere with the current work of the PE and may be done concurrently. After the creation of $\kappa_j(r)$, the integrated interface is operated by t to control the local work on j according to the received requests. The interface should therefore allow to start and stop working on the job in a preemptive way. The idea to use a single control thread per PE is based on the limited support for message passing to and from multiple threads of a single process. The decision has the additional effect that the individual methods in the interface are mutually exclusive by design.

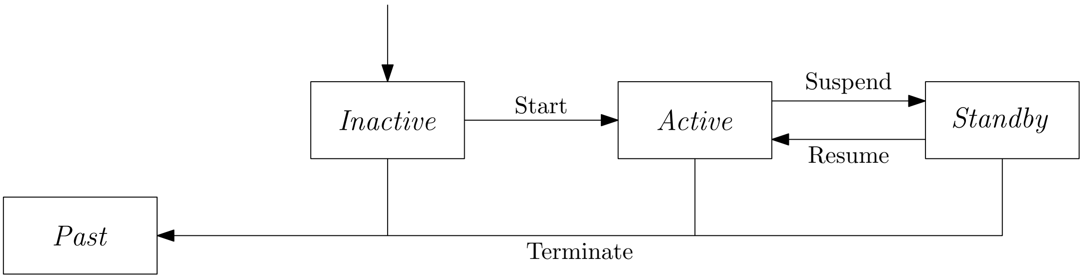


Figure 3.4: A state diagram that shows the lifecycle of a job on a processing element.

The lifecycle of a job in a PE is shown by the state diagram in Fig. 3.4. We call $\sigma(\kappa_j(r))$ the current state of the context of job j on PE r. In the following list we describe what the system expects if $\sigma(\kappa_j(r))$ equals the given state.

Inactive: This states represents a newly created job context $\kappa_j(r)$. This means r has not worked on j in the past. It should therefore not contain any solving resources and active threads.

Active: Only a single job context may be in this state at a time per PE. If this is the case for job j for PE r, it means that r is in the process of creating solving resources in $\kappa_j(r)$ or it already uses them to work with its full computational power on j.

Inactive: A job j in this state was previously processed by r. The PE was then scheduled to suspend its work on j. The system now assumes that $\kappa_j(r)$ contains solving resources and thus that r can easily resume to work on j.

Past: If a job j leaves the system, each κ_j is put into this state by the corresponding controlling thread. This state represents that all created resources for solving are cleaned up or in the process to be cleaned up. This eventually allows the PE to forget job j by deleting κ_j.

The four methods on the edges of Fig. 3.4 control a job's state and therefore are members of the job interface and correspond directly to a job request. We explain them in the following list. Each of them is supposed to simply initiate the state transition and therefore to return instantly. If this transition requires heavy work in the job's context, a separate thread should be started to perform this concurrently.

start: After $\kappa_j(r)$ was created, $\kappa_j(r).start_M$ may be called by t to adopt the job j. The method leads to the creation of a solving engine in the corresponding job context. The solving engine then starts one or multiple solver threads that do actual work on j. The number of started threads is given by a global parameter. In practice this parameter is set to the number of cores per PE.

If this methods returns, the PE enters the job tree T_j and $\sigma(\kappa_j(r))$ is set to *Active*. The references of the new job node and its parent are automatically updated.

The work on a new job can only be started by a PE r if $\forall j \in r\colon \sigma(\kappa_j(r)) \neq Active$. This is guaranteed by the controlling thread of r.

suspend: If r is supposed to suspend the work on its current job j with $\sigma(\kappa_j(r)) = Active$, $\kappa_j(r).suspend_M$ is called. This does not mean that job j is no longer an active job in the system, but that the distributed load balancing computation has forced a rescheduling. The PE r is therefore supposed to stop all solver threads in $\kappa_j(r)$. This can be done by by forcing them to fall asleep or by interrupting their solving and joining them. Among others, the following considerations should be taken into account:

1. If the threads remain in the suspended state and never resume their solving, is the solving on j in total still correct?
2. How easily and quickly can the threads continue to work on j if they are resumed?

Furthermore, if it is required for correctness, the control thread t may send a last parting message to T_j during the call of $\kappa_j(r).suspend_M$. This can be used to share any locally reserved work.

If this method returns, all usage of computational resources in r on j are in the process of being stopped. Therefore, r is removed from T_j and $\sigma(\kappa_j(r))$ is set to *Standby*.

resume: If a PE r should resume working on a formerly active job j, $\kappa_j(r).resume_M$ is called. Similar to $start_M$, the controlling thread t of r guarantees that r does not have an *Active* job. It is also guaranteed that $\sigma(\kappa_j(r)) = Standby$.

If this method returns, r is assumed to have resumed working on j. It is therefore reinserted in T_j and $\sigma(\kappa_j(r))$ is set to *Active*. The references in the resumed job node and its parent are then automatically updated.

terminate: The last method $terminate_M$ is called on κ_j, when a job j leaves the system. A job leaves the system if its task was solved by some PE or if it was aborted due to a timeout. The method is therefore called eventually for every job context, regardless of its current state.

On a call of $\kappa_j(r).\mathit{terminate}_M$, the addressed PE r is prompted to clean up all solving resources in $\kappa_j(r)$. This should allow the control thread t to effortlessly delete the job's context at a future point in time.

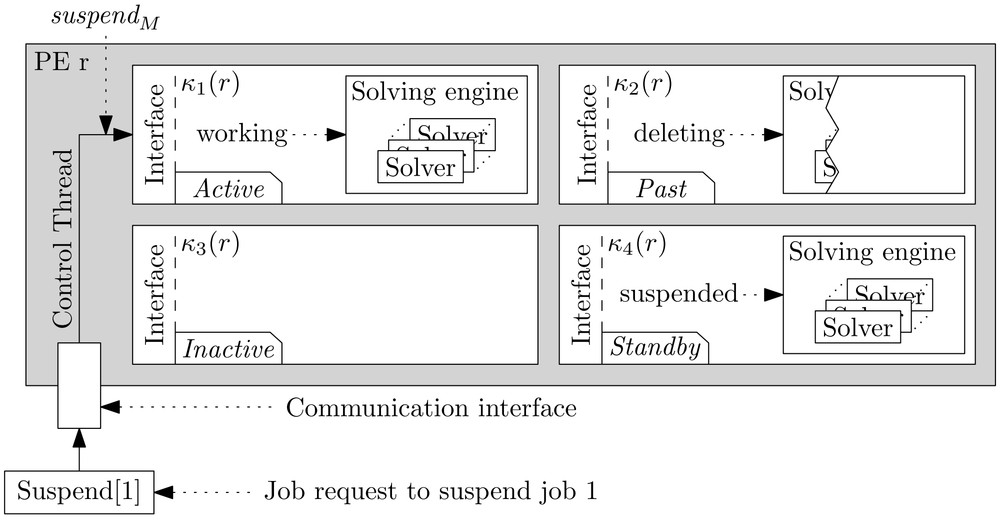


Figure 3.5: A schematic illustration of a PE with four job contexts in various states. The control thread currently reacts to a suspension request by calling the appropriate method on the corresponding context.

In Fig. 3.5 we illustrate a PE r that contains four job contexts $\kappa_1(r), \kappa_2(r), \kappa_3(r)$ and $\kappa_4(r)$, each in a different state. The context of job 1 is working. Job 2 currently deletes its solving engine. Job 3 is newly created and therefore has no solving engine. Job 4 was suspended and therefore has an inactive solving engine. The control threads is in the processes of suspending $\kappa_1(r)$ because it received a message to do so.

In addition to these lifecycle controlling methods, there are also methods in the job interface that are supposed to report on the job's status on a PE. They are used for communication between the solving engine in a job context and the control thread t. We will name and explain all necessary methods in the following list:

solved: This method is called periodically on each job node. It is used to report the solving status of job j to the associated control thread. A successful solving of j is broadcast in T_j to allow each PE to terminate the job.

demand: This method is only called in the root node. It returns the job's current resource requirements for the purpose of load balancing.

destructable: This method is called if $\sigma(\kappa_j(r)) = \mathit{Past}$. It allows the control thread of r to query for the progress of the deletion of the solving engine in $\kappa_j(r)$.

For collaborative solving, the interface also provides a way for communication between multiple job contexts that work on the same job. This is implemented by the controlling thread of r regularly asking whether $\kappa_j(r)$ wants to send a job-specific message. This is done via the method $\mathit{startComm}_M$. In it, the solving engine in $\kappa_j(r)$ can then address a PE in T_j via its stored references and send messages through the controlling thread. Sending messages over T_j guarantees that a job-specific message exclusively reaches PEs that have completed *start*$_M$ and thus have a solving engine in their context which can handle the message. To handle received messages *handleComm*$_M$ needs to be implemented. This method is called in $\kappa_j(r)$ of the addressed PE r and passes the received message to its the solving engine. This invocation happens regardless of $\sigma(\kappa_j(r))$. The call to *handleComm*$_M$ on $\kappa_j(r)$ can additionally be used to send further messages from r.

3.3 Mallob in the domain of SAT solving

The Mallob platform has its place in the domain of SAT solving after winning the Cloud Track of the International SAT Competition 2020 using a malleable implementation based on HordeSat [3]. However, since the competition was not aimed at solving multiple SAT problems simultaneously, the system was configured to process the problems sequentially using the maximum number of PEs for each problem from the beginning.

In the context of SAT solving on Mallob, each job represents an instance of the SAT problem. The corresponding formula is therefore shared with each assigned PE as the task of the job. In the implementation based on HordeSat, the interface is designed to start multiple solver threads on job adoption. The solver threads are all working on the problem's formula using a globally uniquely configured instance of the sequential SAT solver Lingeling [6]. In addition, learnt clauses are extracted from these threads and accumulated in the solving engine. The engines then use the interface's communication mechanism to periodically share these clauses with all PEs that work on the same job by performing an all-to-all clause exchange. This form of exchange can be produced by aggregating all extracted clauses to the root node and then broadcasting the reduced set of clauses back to each job node like shown in Fig 3.3. The aggregation step also includes a clauses filtering step in which duplicate clauses are removed. The received clauses are then incorporated into the solving process of the local solver threads. To enable the preemption capabilities of jobs, all parallelization in the job context is done using child processes. This allows straightforward suspension, resumption and termination on the OS level through signals with the minor overhead of creating additional processes.

4 Design Decisions

The main focus of each of our approaches was to form a SAT solving method using search-space splitting that allows for malleability and works well on problem instances from various applications. We did not restrict ourselves to problems which are known to be highly complex or to work well with Cube&Conquer [17, 21]. Because of this decision we based all our concepts on a generic state-of-the-art CDCL solver at its core. We did this by building them around the incremental interface IPASIR presented in Subsection 2.3.4. The reason for this decision is that modern CDCL solvers dominate in the discipline of solving diverse problem instances that originate from practical contexts. It also contributes to the conceptual simplicity of our approaches. The sophisticated internals of modern solvers which include specialized inprocessing steps are not regarded. The solver is simply used through the IPASIR interface as a blackbox.

Secondly, we decided to design our algorithms as unintrusive as possible regarding the sequential SAT solvers. In particular, we aim to avoid frequent interruptions to their solving process. Interruptions are made only when necessary to ensure correctness or when required for malleability. We based this decision on these two assumptions:

1. In previous testing we observed that frequently forcing a CDCL solver to exit its internal solving loop via $terminate_I$ has negative effects on its overall performance. This may be even worse if we change its configuration during the interruption. One reason for this might be that a solver may be interrupted during a subroutine and is unable to resume from exactly where it left off and therefore cannot finish an inprocessing step. Another reason for this behavior could be that made solving progress is discarded to return the solver in a universal state.

   Also, interrupting a solver that could have been close to solving a major (sub)-problem, only to reconfigure it, would degrade our algorithm. Especially given the distributed context with an abundance of processing resources, we decided to allow redundancy to increase the performance of our method in these cases.

2. Allowing for malleability will inevitably force solvers to be stopped because almost each associated processing element could be rescheduled to another job at any time. If we combine this condition with an algorithm that also requires solvers to be interrupted, we end up with frequent interruptions and therefore with an algorithm that will not perform well.

   If for example a currently working solver s_1 needs to be interrupted to allow the new solvers s_2 and s_3 to work and if s_2 and s_3 leave the job again shortly thereafter, s_1

was interrupted unnecessarily. Instead, we focused on an approach that allows for additional solvers to participate without impairing the current solvers.

4.1 On finding splitting literals

A key component in a parallel SAT solver that is based on the search-space splitting approach is to find a good method for finding well suited splitting literals. This is very important because the quality of the splits is crucial for the resulting overall performance. This trait is explained in more detail in the Subsection 2.3.6. In our search for a good method we focused on the following three characteristics:

- Does the split result in two sub-formulas that are individually easier to solve than the original formula and comparably equally difficult to solve?
- How time-consuming is the process?
- Is it possible to enter learned information and thus to make split decisions using a currently active solving progress?

The most intuitive option is to utilize a modified version of a state-of-the-art lookahead solver. This idea originates from the work that coined the term Cube&Conquer [18]. A lookahead solver is expected to generate good splitting literals because of the sophisticated heuristics it uses. A candidate for this is the lookahead solver March [19]. It is used as an option to generate cubes in the state-of-the-art distributed Cube&Conquer solver Paracooba [17]. Because March is designed as a standalone command-line tool, Paracooba starts it by using a separate OS process and then collects the cubes that were written onto the hard disk. This workflow can be adapted if we also want to create a fixed size of cubes. However, if we want to perform the lookahead procedure at different points in time using the current solving status, its integration is very unpractical because we cannot reuse a solver instance and must incur the additional overhead required to make this system call each time. Another reason against March is that the version of Paracooba which relied on it did not perform very well in the 2020 SAT Competition. The reason for this most likely is that March is designed to find promising decision variables based on specialized heuristics, but is in doing so nowhere near as computationally efficient as a state-of-the-art CDCL solver. This makes its application unsuited for a time critical SAT solving platform.

Fortunately, multiple modern CDCL based SAT solvers include the lookahead procedure. For example the solver Cadical includes one which is "in comparison to March less tuned" [13]. This allows us to use the lookahead procedure in combination with state-of-the-art incremental SAT solving features like reducing the complexity of the formula by adding good redundant clauses or assuming a partial assignment without having to use a fresh solver instance. In addition, the version of Paracooba that relied on Cadical to generate cubes performed much better in 2020 SAT Competition. This could be caused by requiring less time to generate cubes because its lookahead procedure profits from the highly optimized data structures of the very recent CDCL solver. We therefore decided to use this way to find splitting literals. It allows us to form our method of generating cubes in a similar way to how we solve them. As we chose the CDCL solver Cadical to generate cubes, we will reference a call to its lookahead procedure using $\textit{lookahead}_C$.

The work on the adaptive parallel SAT solver Ampharos [1] presents another way for finding splitting literals using a lookback heuristic. Instead of using a complicated lookahead procedure, CDCL solvers are used to solve a problem until a set amount of conflicts have occurred. Then the solver is interrupted and the literal that would have been assigned at the start of the next recursion is returned. This procedure can be used to quickly find a good splitting literal of a problem that has been proven to be complicated for a CDCL solver. However, we did not consider this approach because it would require to interrupt solvers. Also, getting the next decision variable is not a typical feature included in most state-of-the-art CDCL solvers and would therefore deviate from the standard interface. In addition, the required thresholds for an interruption, such as the number of conflicts or number of decisions, are difficult to set correctly because they scale with the structure of the formula.

5 First Approach: Static Cube&Conquer

In this section we explain our first method to port parallel SAT solving using the search-space splitting approach to the Mallob platform. We call this method *Static Cube&Conquer* because we use a lookahead procedure to generate cubes once in advance as explained in Subsection 2.3.7. We present our approach by first giving an overview of the concept and explaining an important used technique. Then we introduce all important actors and explain how they allow for malleability.

5.1 The concept

Cube&Conquer in its simplest form is based on the following two subsequent steps:

1. Partition the search space into many sub-problems called cubes by using a lookahead procedure to find good splitting variables.
2. Solve all cubes using concurrent CDCL solvers. If one is proven to be satisfiable, return SAT. To determine UNSAT, all cubes need to be proven unsatisfiable.

We realize this algorithm using the manager-worker principle. The manager is responsible for creating, managing and distributing the set of cubes C for the formula F_j of job j. It is therefore active during step one and two. The workers are activated with the beginning of step two. They then request the manager for cubes and solve the corresponding sub-formulas. If a worker finds a satisfying assignment for a cube, it can generate a valid assignment for the entire formula. This allows the worker to individually terminate the algorithm and return SAT. Otherwise, the worker sends its results back to the manager and thus requests new cubes. The manager terminates the algorithm and returns UNSAT if it determines every cube to be unsatisfiable based on all received worker results.

This concept may be effortlessly transferred to job trees in Mallob. The root node $p_0(j)$ holds the manager. The other nodes $p_1(j), p_2(j), p_3(j), \dots$ hold the workers. As the manager only has light computing tasks, $p_0(j)$ may host a worker as well. The requesting of cubes and sending back results is done via the message passing system that Mallob offers. The mapping of this concept to a job tree is illustrated in Fig. 5.1.

5.2 Pruning

To define the results of a worker and to allow the manager to use them to mark certain cubes unsatisfiable, we used the pruning technique from [1]. Every solver in each worker

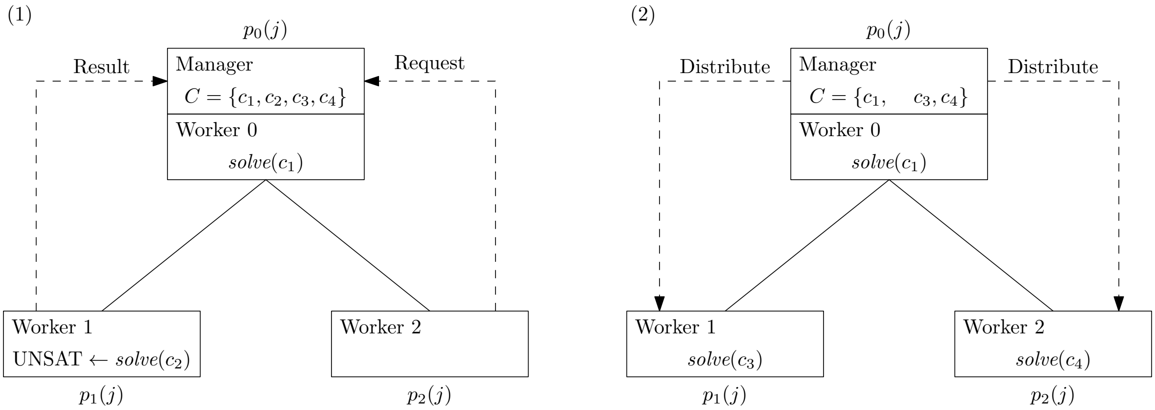


Figure 5.1: Illustration of the concept of this approach mapped onto a job tree consisting of three PEs. The cubes in the manager were created as depicted in the example from Subsection 2.3.6.

processes a sub-formula of F_j that is defined by a given cube $c_i \in C$ by assuming the set of literals $\{l \mid l \in c_i\}$ with *assume*$_I$ and then executing *solve*$_I$. If this returns UNSAT it means that the the formula $\bigwedge_{l \in c_i} l \wedge F_j$ was proven to be unsatisfiable by the used solver. The worker then extracts the assumed literals used in this proof $\phi_i \subseteq c_i$ by calling *failed*$_I$. These responsible assumptions are in the following referred to as *failed assumptions*. This allows us to generalize the proof of unsatisfiability to the potentially less restricted formula $\bigwedge_{l \in \phi_i} l \wedge F_j$. The failed assumptions are then sent to the manager, allowing it to *prune* all cubes in C that contain ϕ_i as a subset. In the occasion of a proof of unsatisfiability for a cube c_i with $\phi_i = \emptyset$, the solver has proven the formula to be unsatisfiable regardless of the given partial assignment. The worker can therefore directly return UNSAT and terminate the algorithm.

5.3 The manager

The role of the manager needs to be assigned at the start of each job. The manager instance of the job j is therefore created on the root node $p_0(j)$ during the first call to *start*$_M$. After its creation, it is tasked to generate the set of cubes C according to a predefined height parameter h. This is done by a single thread that uses *lookahead*$_C$ and *assume*$_I$ in an interleaved way. This results in a perfect binary tree with height h and 2^h leaf nodes which represent the set of cubes $C = \{c_i \mid i \in \mathbb{Z}, 1 \leq i \leq 2^h\}$, similarly to the example in Subsection 2.3.6. This means that the lookahead procedure needs to be called 2^{h-1} times in total, once for each inner node. The assume functionality is used here to choose each next splitting variable under the partial assignment given by the path leading to it. This allows us to reuse the same solver instance to generate all cubes.

During the cube generation the root node sets the demand of its job j to $d_j = 1$. This is done to prevent the allocation of additional PEs while there is no work to be distributed.

After the cubes are generated, the root node creates and starts a local worker and raises the demand by using the default growth function.

5.3.1 Workflow of the manager

The workflow of the manager during the second phase consists only of interpreting messages and responding accordingly. There are two types of messages in this approach: (1) a message to return any failed assumptions to the manager and simultaneously request new cubes and (2) a response to a worker which contains the requested unsolved cubes. The amount of cubes sent is defined by the minimum of a parameter *cubes-per-worker* and the number of remaining unsolved cubes. The cubes are taken from the beginning of a sorted queue. They are sorted in ascending order by the number of times they were distributed to a worker.

If the manager receives a request it proceeds with the following three step procedure:

1. If the request contains a set of failed assumptions Φ, use it to prune all cubes $c \in C$ with $\phi \subseteq c, \phi \in \Phi$. If this results in the removal of all remaining cubes, the manager marks the job as unsatisfiable.
2. Take the $\min\{\textit{cubes-per-worker}, \text{size}(C)\}$ first entries of the priority queue which is used to store C and send them to the requesting worker in a response. This includes sending an empty set if $C = \emptyset$.
3. Increment the value of "times distributed" for each sent cube and re-insert them into the queue.

If there are no messages, the manager is idle.

Because it resides on the root node, the manager is never paused using $\textit{suspend}_M$. It may however be interrupted. If $\textit{terminate}_M$ is called during the first step, the cube generation thread is interrupted via $\textit{terminate}_I$. The second step is then never reached. If $\textit{terminate}_M$ is called during the second step, the manager can simply be destroyed. The manager does not need any dedicated threads to distribute cubes. All of its logic is done by the controlling thread in $\textit{handleComm}_M$.

5.3.2 Completeness

The completeness of this parallel SAT solving method is based on keeping each unsolved cube in the manager and on only pruning them when a matching set of failed assumptions is received. This makes each distribution of cubes to a worker a non-binding work assignment. A designated worker may therefore leave a job at any time to allow for load balancing without influencing C. In the worst case scenario with a job j of infinite duration and strongly fluctuating load balancing such that each job node $p_x(j), x > 0$ never is able to solve a cube, the completeness follows from each cube in C eventually being distributed to the worker on $p_0(j)$ which will solve each cube because it uses a complete solver.

5.3.3 Randomization

In the standard configuration, all cubes are initially placed in the priority queue ordered from the left-most leaf to the right-most. This places cubes representing similar partial assignments adjacent. This can be seen as an advantage because similar cubes are likely to be distributed to the same worker, allowing it to work on more similar problems and thus making more use out of the incremental SAT solving feature. However, it can also be seen as a disadvantage because if the number of cubes greatly exceeds the number of workers, all solving is done in similar sub-spaces. The work on more diverse search-spaces is postponed. In addition, this approach introduces an order between all cubes in which they are processed by workers. This results in the following problem: If there are two cubes c and c' with c before c' and c being very hard to solve and c' being very easy, every worker that receives c and c' gets stuck on c.

To test this assumption, we added an option to randomize the distributed cubes. This is done by initially placing the cubes into the queue in a random order and by reshuffling each distributed set of cubes.

5.4 The worker

In this section we discuss the workflow of a worker. However, to do this, we must first introduce an extension to the IPASIR interface.

5.4.1 Extending IPASIR to suspend solvers

As explained in Section 3.2 each processing element in Mallob is required to be able to pause its active work on a call of $suspend_M$. Since we will be using IPASIR-based solvers in each worker, we require a method to expand this interface to produce this desired mechanism. This should also be done respecting our design decisions by being minimally invasive. We achieved this by inheriting an older concept from the implementation based on HordeSat in Mallob. The concept is based on installing a monitor into the main loop of the solver by placing it into the predicate which is inserted via $terminate_I$. The definition of the interface requests the solver to periodically check the predicate to allow it to indicate termination. During such a check, we can then use the monitor to halt the solving thread. The now sleeping thread is then stored in the queue of a condition variable and waits for a notification to resume. With this method, we can pause the execution of a solver that implements the IPASIR interface without altering its workflow by allowing it to continue exactly where it left off.

5.4.2 Workflow of the worker

If the second phase of a job j is reached, each job node $p_x(j)$ will instantiate the worker in its solving engine in κ_j. The PE of $p_0(j)$ does this at the end of its cube generation, the PEs of the other job nodes $p_x(j), x > 0$ do so during the call to $start_M$. A worker instance contains one or multiple cube solver threads s_n with $n \in \mathbb{N}$ that start working as soon as they are created. Each of them encapsulates an instance of an IPASIR based solver which the thread uses to sequentially process all passed cubes in a loop. If new cubes are required to continue, the thread stores all extracted failed assumptions in the wrapping worker instance and informs it to request new cubes. This instruction is then asynchronously read by the controlling thread of the PE during a periodic call to $\mathit{startComm}_M$ which then sends the accumulated failed assumptions to the manager and requests new cubes. The received unsolved cubes from the manager are then passed to the waiting solver thread during a call to $\mathit{handleComm}_M$ so that it can continue its solving loop. To reduce overhead, a single request is sent for all waiting threads.

If a PE r is supposed to leave the job j with $\sigma(\kappa_j(r)) = \mathit{Active}$, $\kappa_j(r).\mathit{suspend}_M$ is called. This means, all cube solver threads are instructed to halt their execution. This is achieved by $\mathit{suspend}_I$ being called on the encapsulated solver instance of each thread. The thread is thus only stopped when it is processing a cube. All communication still needs to be completed. This is necessary to leave the thread in an easily resumable state. If resume_M is called subsequently, the thread simply resumes processing the cube.

There are three scenarios that lead to the termination of a solver thread.

- $\mathit{terminate}_M$ is called while the thread is currently processing a cube or suspended.
- A solver found a satisfying assignment. The job is then also marked as satisfiable. The control thread is notified of the successful solve during its next call to solved_M.
- An empty set of cubes is received because the manager was able to prune all cubes in C.

The solving loop of a solver thread and its communication with the control thread is depicted in Fig. 5.2.

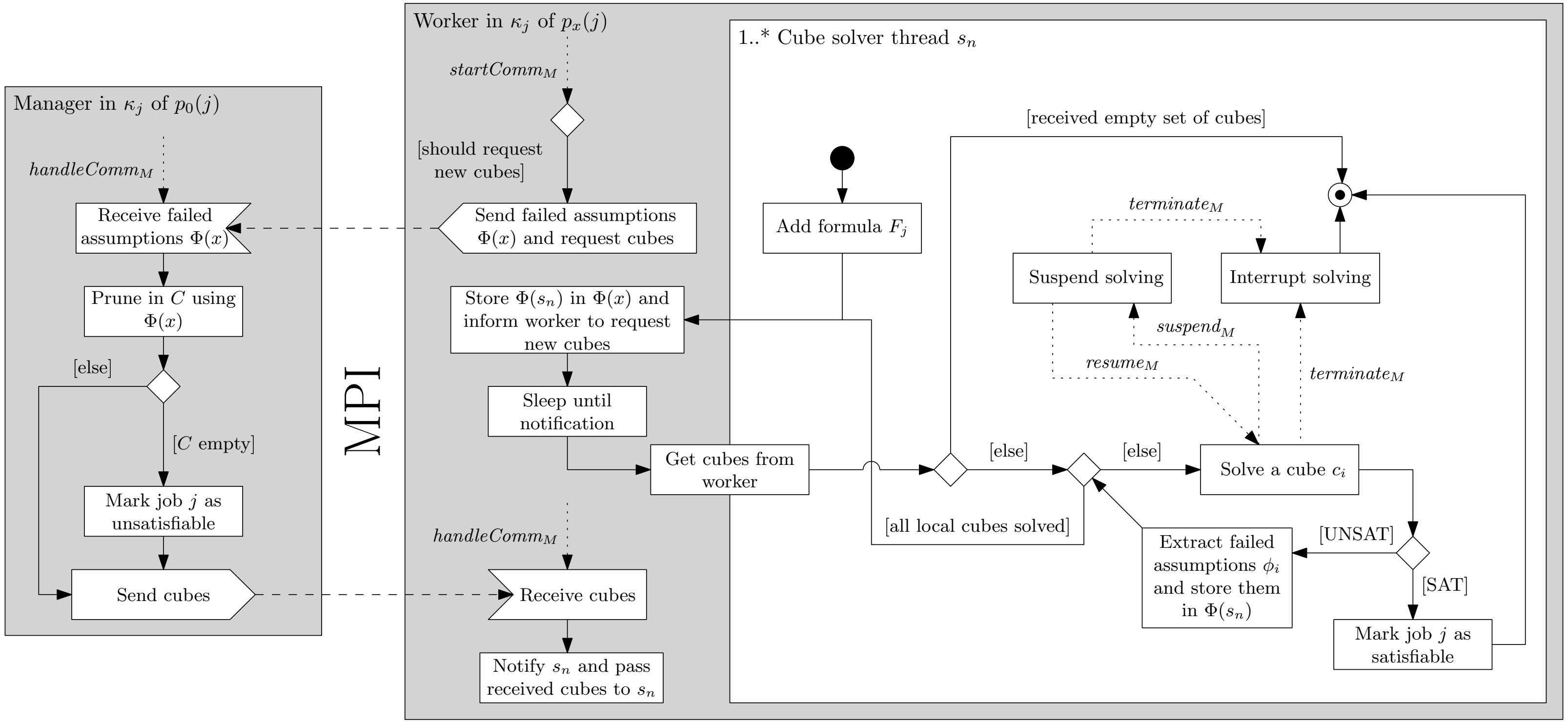

Manager in κ_j of $p_0(j)$
$handleComm_M$
Receive failed assumptions $\Phi(x)$
Prune in C using $\Phi(x)$
[else]
[C empty]
Mark job j as unsatisfiable
Send cubes
MPI
Worker in κ_j of $p_x(j)$
$startComm_M$
[should request new cubes]
Send failed assumptions $\Phi(x)$ and request cubes
Store $\Phi(s_n)$ in $\Phi(x)$ and inform worker to request new cubes
Sleep until notification
Get cubes from worker
$handleComm_M$
Receive cubes
Notify s_n and pass received cubes to s_n
1..* Cube solver thread s_n
[received empty set of cubes]
Add formula F_j
$terminate_M$
Suspend solving
Interrupt solving
$suspend_M$
$resume_M$
$terminate_M$
[else]
[else]
Solve a cube c_i
[all local cubes solved]
Extract failed assumptions ϕ_i and store them in $\Phi(s_n)$
[UNSAT]
[SAT]
Mark job j as satisfiable

Figure 5.2: Illustration of the workflow of a cube solver thread in a worker instance and its communication with the manager. Each smaller rectangle represents an action, each edge denotes a transition following the completion of the action. Dotted edges denote the start of concurrent actions or transitions to and from unfinished actions, in case of preemption. All of these cases are dependant on an asynchronous call of a method of the Mallob interface. Dashed edges represent send messages connecting their send and receive action. Diamond shapes represent decisions.

6 Second Approach: Dynamic Cube&Conquer

The initial cube generation plays a major role in the previous approach. It enables a centralized work distribution strategy that makes workers independent of each other. This independence allows for a straightforward integration of malleability because each additional worker of job j in $p_x(j), x > 0$ is not required for completeness and can therefore be discarded at any time. However, it is based on an initial step that does not make use of the massively parallel environment of Mallob. We therefore created a second more decentralized approach that uses parallelism from the beginning. This approach is partly inspired by the Divide&Conquer technique explained in Subsection 2.3.8.

The key idea of this second approach is to start solving as soon as the job is entered into the system. This includes trying to solve the unmodified formula. Cubes are generated concurrently to provide diversified and specialized work for additional solvers. New cubes are created in unsolved search-spaces and their generation involves knowledge of the current solving process. This however leads to some redundancy, because solvers that work in a non-disjoint search-space to a new cube are not interrupted. To prevent complete redundancy between two active solvers, the cube generation and distribution strategy is designed in a way that disallows the generation of duplicate cubes. Found failed assumptions are spread to all solvers similarly like learnt clauses in the portfolios approach.

By doing so we combine ideas of the portfolios and the search-space splitting approach. We support the solving process of each solver by sharing failing assumptions. Each solver works on a sub-space to allow more diverse solving.

We keep the name Cube&Conquer for this approach, because we continue to refrain from interrupting solvers to find splitting variables. We will instead rely on separate solver instances that are fed with accumulated information to generate cubes using the lookahead procedure $lookahead_C$.

6.1 The concept

We created this approach by defining multiple linked invariants per job node. This allows for nearly the same functionally per node and thus making the general method much simpler to understand and easier to control. The invariants are based on each job node possessing cube solving and cube generating capabilities. Both capabilities consist of

IPASIR-based solver instances in separate threads. They are supported by communication protocols over the job tree. We first introduce how cubes are shared between nodes and how new cubes are generated. We then go into how each node uses its solving capabilities to work on cubes, and how the solving results are distributed in the job tree and used by the receiving node. Finally we specify each termination scenario of the resulting algorithm.

Each available cube of a job j is uniquely assigned to the job node it is contained in. If a node $p_x(j), x > 0$ does not hold any cubes, it relies on its parent node to obtain cubes. This is done by $p_x(j)$ sending a request q_x for cubes to its parent via the communication mechanism of Mallob. The parent is then supposed to fulfill q_x by passing cubes to $p_x(j)$. To pass cubes, they must be removed locally and then assigned to the requesting node by sending them in response. This is done to keep the condition of each cube being uniquely assigned. If the parent is unable to do so, the request of $p_x(j)$ is forwarded to the next parent node and so forth, similar to the aggregate operation. Eventually, a request reaches the root node $p_0(j)$ and resides there until the root fulfills it.

If the root node does not contain any cubes, it deviates from this requesting chain because it relies on its children to obtain cubes. This is done by $p_0(j)$ sending a request for cubes to its direct children $p_1(j)$ and $p_2(j)$. If a child can fulfill this request, it passes cubes in the same manner as in the protocol above. Is a node $p_x(j)$ unable to do so, it forwards the requests to its children $p_{2x+1}(j)$ and $p_{2x+2}(j)$. If $p_x(j)$ is a leaf and cannot fulfill the request, it is discarded.

As long as there are locally assigned cubes, a node may split them further using its generating capabilities to provide work to the local solving capabilities. This is done by taking a local cube c_i and using $assume_I(c_i)$ followed by $lookahead_C$ to find a literal l_{i+1} with $l_{i+1} \notin c_i$, creating the two new cubes $c_{2i+1} = c_i \wedge l_{i+1}$ and $c_{2i+2} = c_i \wedge \neg l_{i+1}$. The extended cube c_i is then removed from the local set of cubes and replaced by c_{2i+1} and c_{2i+2}. The indices of the cubes describe their position in a binary tree generated by repeated application of this extension procedure, starting from the empty cube c_0 that represents the unrestricted formula. In combination with the requirement that each cube is uniquely assigned to a job node, this extension protocol disallows the creation of duplicate cubes. This follows from the deductions below:

- Each job node $p_x(j)$ contains a pair-wise disjoint subset C_x of the leaves of the binary cube tree. Inner cube nodes are never contained in a job node because they are discarded after their extension. This is illustrated in Fig. 6.1.
- Any pair of cubes $\{c_a, c_b\}$ with $c_a \neq c_b$ from any two subsets $c_a \in C_x$ and $c_b \in C_y$ always differ in the polarity of one literal l_e with $l_e \in c_a$ and $\neg l_e \in c_b$. This follows from the fact that both are different leaves in the binary tree of expanded cubes, which forces the path to each of them from the root node to split at an inner cube c_{e-1}.
- We can then apply the extension procedure on c_a and c_b to generate the set of four new cubes $\{c_{2a+1}, c_{2a+2}, c_{2b+1}, c_{2b+2}\}$. The two pairs $\{c_{2a+1}, c_{2a+2}\}$ and $\{c_{2b+1}, c_{2b+2}\}$ still differ in l_e. The two cubes in both pairs differ in the polarity of the new literal l_{a+1} or

l_{b+1} respectively that was found during the extension procedure. This makes each newly generated cube unique.

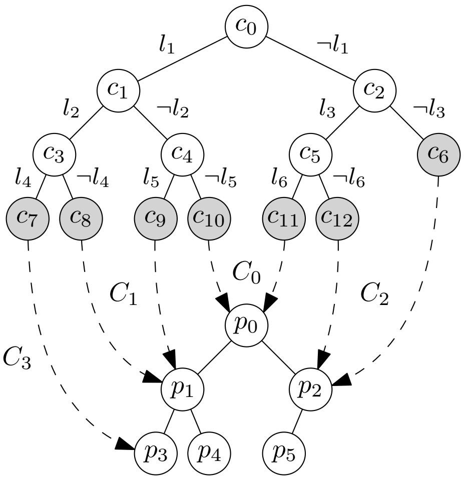


Figure 6.1: Illustration of the creation of a currently active set of cubes $\{c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}\}$ and how they are assigned the job nodes p_0, p_1, p_2 and p_3. The job nodes p_4 and p_5 do not hold any cubes.

The solving capabilities of each node $p_x(j)$ are used to concurrently work on a subset of the local cubes C_x using one or multiple cube solver threads. If the formula F_j of the job j was proven to be unsatisfiable under the partial assignment of a cube c_i, the failed assumptions ϕ_i are extracted using *failed*$_I$. If $\phi_i \neq \emptyset$, the failed assumptions ϕ_i are transformed into the failed clause ψ_i and stored locally. A *failed clause* $\psi_i = \bigvee_{l \in \phi_i} \neg l$ is a clause that consists of the inverse literals in ϕ_i. These locally accumulated failed clauses are then all-to-all exchanged using the explained aggregate and broadcast operations of the job tree. If a job node receives new failed clause via the broadcast operation, the clause is added to each solver instance in both capabilities using *add*$_I$. This allows us to share the knowledge of the generated proof of unsatisfiability for the partial assignment given by ϕ_i to other solver instances via incremental SAT solving. Doing so should therefore simplify F_j for each solver instance and thus improve its solving and cube generating capabilities. To prevent unnecessary additions, each node holds a filter which it uses to detect and discard newly received but already added failed clauses.

There are two termination scenarios for this approach. If a solver finds a satisfying assignment while processing any cube, it may terminate the job and return SAT. If a solver proves the formula F_j of the job j to be unsatisfiable under a cube c_i with $\phi_i = \emptyset$, UNSAT is returned and the algorithm is terminated. Because of the sharing of failed clauses, the UNSAT case also accounts for the circumstance that the formula was distributedly proven to be unsatisfiable for a set Φ of partial assignments with $\bigvee_{\phi \in \Phi} = F_j$. Such a case is illustrated in Fig. 6.2. Note that $\phi_i \subseteq c_i$. Another way to recognize this is when a solver proves a cube unsatisfiable because its added failed clauses are inherently unsatisfiable. The concept therefore allows the use of the search-space splitting approach without requiring a designated manager with global knowledge of all generated and solved cubes.

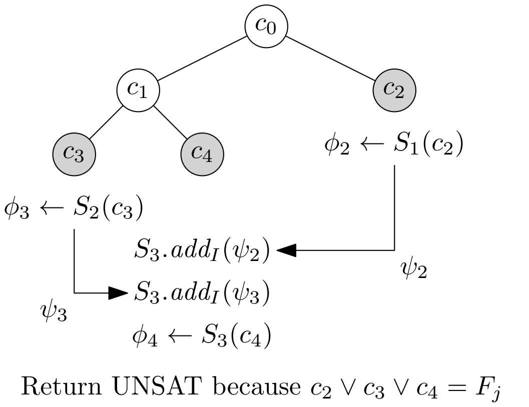

c_0
c_1
c_2
c_3
c_4
$\phi_2 \leftarrow S_1(c_2)$
$\phi_3 \leftarrow S_2(c_3)$
$S_3.add_I(\psi_2)$
$S_3.add_I(\psi_3)$
ψ_2
ψ_3
$\phi_4 \leftarrow S_3(c_4)$
Return UNSAT because $c_2 \vee c_3 \vee c_4 = F_j$

Figure 6.2: Illustration of a scenario where the sharing of failed clauses allows a solver S_3 to conclude that all leaf cubes were proven to be unsatisfiable. S_1 is a solver who proves c_2 to be unsatisfiable and then sends ψ_2 to S_3. The solver S_2 does the same for c_3. S_3 starts to process c_4 after S_1 and S_2 have finished their work and it has added their failed clauses. If S_3 then proves F_j to be unsatisfiable under c_4, it can return UNSAT and terminate the work on j.

In Fig. 6.3 we illustrate all invariants and multiple communication protocols for the job node p_2 from Fig. 6.1. In the first step, the solving capabilities begin to process the local cube c_{12}, while at the same time the generating capabilities start extending the local cube c_6. In step two, c_{12} was proven to be unsatisfiable, generating the failed clause ψ_{12}. The cube c_6 was extended and thus replaced by the new cubes c_{13} and c_{14}. Simultaneously, a request for cubes q_5 was received from the child node p_5. Consequently, step three shows p_2 fulfilling the request by sending the cube c_{14} to p_5. In step four, p_2 is instructed to aggregate failed clauses by obtaining an empty set of failed clauses from p_5. It does this by unifying the locally stored failed clauses with the received empty set and sending the resulting set $\{\psi_{12}\}$ to its parent. In the last step, the aggregated set of failed clauses $\{\psi_8, \psi_{12}\}$ is received. After a local filtering, they are incorporated into both capabilities and then forwarded unfiltered to the single child node p_5.

In the following we will introduce how we ported this concept to Mallob. We will do this by introducing multiple used concepts and components and how they interact with each other. We will thereby also go over how this approach allows for malleability.

6.2 Dynamic cubes

To enable the simultaneous solving and expanding of a single cube in a job node, we introduce the concept of dynamic cubes. A *dynamic cube* d_i wraps a globally unique cube c_i and manages its assignment to the local cube processing threads. Each job node $p_x(j)$ of job j therefore implicitly contains the set D_x of dynamic cubes for its local set of cubes C_x. Each dynamic cube d_i can be simultaneously assigned to a single cube solver thread that solves c_i and to the generator thread that performs the extension procedure on c_i. The concept therefore allows both capabilities of $p_x(j)$ to work on a single local cube c_i at

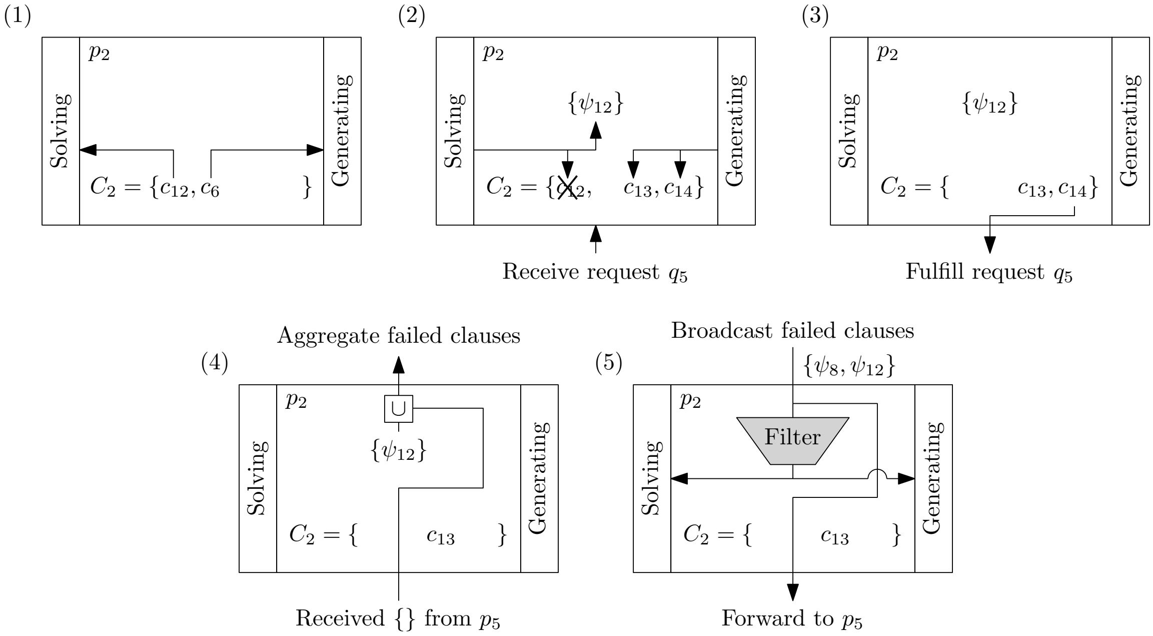


Figure 6.3: Depiction of all invariants and multiple communication protocols for the job node p_2 from Fig. 6.1.

the same time. The assignments of a dynamic cube also restricts its ability to be passed to other job nodes. Only a *free* dynamic cube which is not assigned to any threads is allowed to be passed to a requesting node.

In the following, we assume a dynamic cube $d_i \in D_x$ which is processed by a cube solver thread s and the generator thread g of $p_x(j)$ and discuss both possible interleavings:

- If g finishes first by finding a splitting literal l_{i+1}, the two new dynamic cubes $d_{2i+1} = d_i \wedge l_{i+1}$ and $d_{2i+2} = d_i \wedge \neg l_{i+1}$ are created and replace d_i in D_x. The solver thread s is not notified of this replacement and keeps working on d_i. Instead, d_{2i+1} or d_{2i+2} is virtually assigned to s. This is done to maintain the condition that the solving capability only works on locally contained cubes. This assignment is correct because s works on a more general version of its assigned cube. The possibly found failed assumptions ϕ_{2i+1} or ϕ_{2i+2} would therefore always be a subset of the assigned cube. We show multiple reassignments in Fig. 6.4.

- If s finishes first by proving the formula F_j to be unsatisfiable under the partial assignment given by d_i, it extracts the failed assumptions ϕ_i and uses them to prune cubes in D_x. This means it removes all dynamic cubes $d \in D_x$ with $\phi_i \subseteq d$, guaranteeing the removal of d_i. When g subsequently completes the extension procedure, it recognizes that d_i is no longer in D_x and discards its results.

All access to the local dynamic collection must be exclusive. Otherwise race-conditions may lead to the duplication of dynamic cubes or their disappearance.

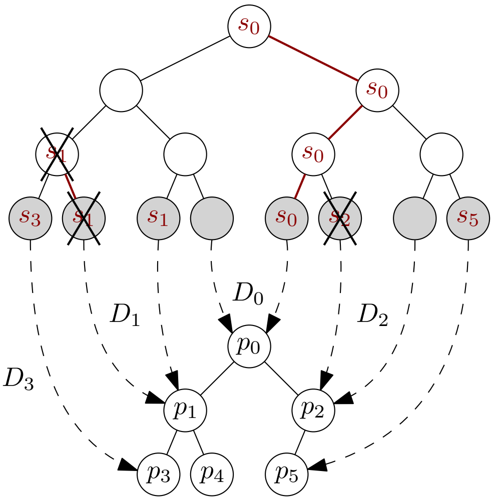


Figure 6.4: Illustration of the reassignments of the cube solver thread s_x from the job node p_x because of a extension of the same cube. Each red edge represents a reassignment. The example is a continuation of the example from Fig. 6.3.

6.3 Completeness

The completeness of this approach relies on the redundancy due to the virtual solver reassignments. This results in the root job node $p_0(j)$ always containing a cube solver thread on the empty cube c_0 and therefore the unmodified formula F_j. The thread is started when the job is adopted and runs as long as the job is in the system. Completeness then follows from the thread internal usage of an instance of a complete solver. All additional work is done to try a complementary approach based on solving sub-spaces.

6.4 The cube solver threads

Each job node contains one or multiple solver threads that work on locally available cubes. The amount is given by the number of cores per processing element. This is done in order to use the full computing power of each PE in solving. All solver threads are created and started during $start_M$. In the following we will go over the solving loop of a cube solver thread s_n from job node $p_x(j)$ which is depicted in Fig. 6.5. We will also go into how s_n affects and is affected by the distribution of the failed clauses. The distribution process is explained in detail in Section 6.5. We will explain how s_n supports malleability in Section 6.8, after covering the generating loop of the generator thread g in Section 6.6.

The first step of s_n is to add the formula F_j to the local instance of an IPASIR based solver. Then the scope of the solver thread is exited and the wrapping solving engine is queried for a cube to work on. This is achieved by requesting exclusive access to D_x. When access is granted and there is a dynamic cube d_i that is not assigned to any solver, assign it to s_n. Otherwise, the thread is put to sleep until a notification is received. A notification for a sleeping solver thread has one of the following three sources and invokes it to retry getting a cube:

1. The local cube generating thread g has successfully extended a local cube.
2. The request for cubes q_x was fulfilled and new cubes were received.
3. $\mathit{suspend}_M$ or $\mathit{terminate}_M$ was called.

After a dynamic cube d_i is assigned to s_n, all in the meantime asynchronously received failed clauses are added to the solver instance from $\Psi(s_n)$. Then the formula F_j is solved under the partial assignment given by c_i. If this results in SAT or UNSAT with $\phi_i = \emptyset$, the job is marked as solved and s_n is terminated. Otherwise, ϕ_i is transformed to ψ_i and stored in $\Psi(x)$. These new failed clauses are then taken into account in the next aggregation. The failed assumptions ϕ_i are also used to prune cubes in D_x. Pruning again requires exclusive access to D_x which is then also used to assign a new cube to s_n, thus repeating the loop.

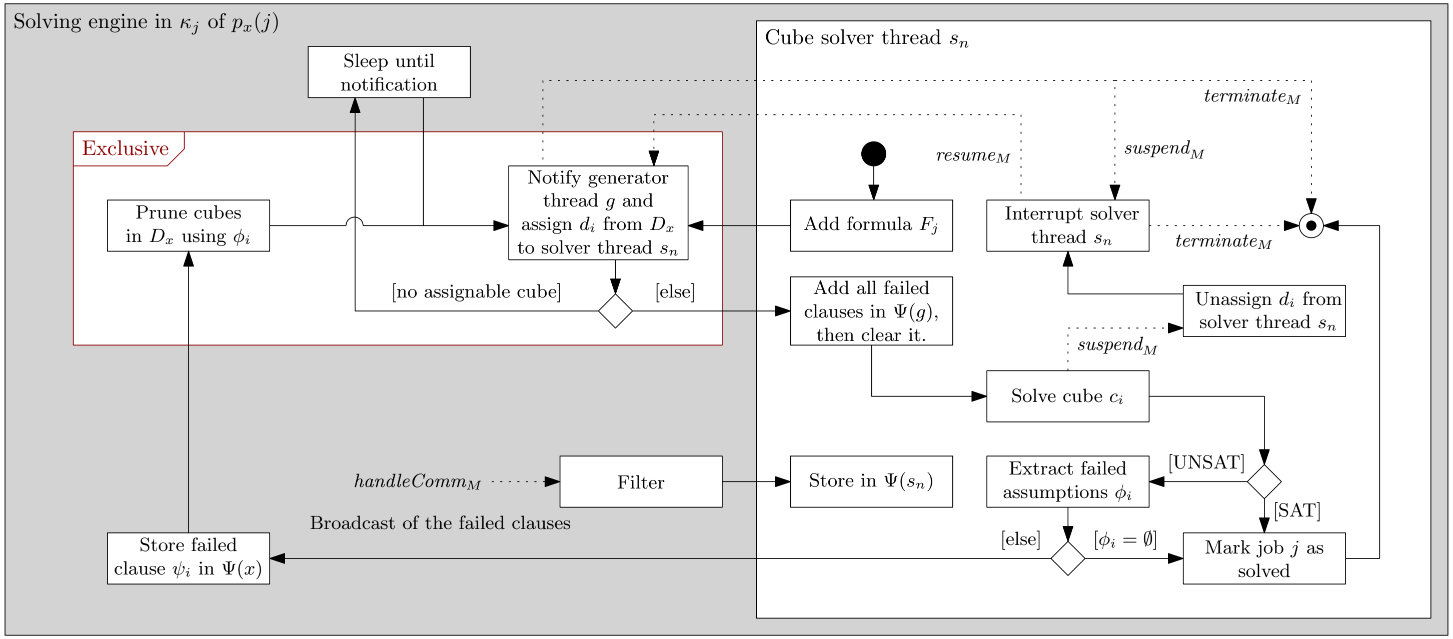

Solving engine in κ_j of $p_x(j)$
Sleep until notification
Exclusive
Prune cubes in D_x using ϕ_i
Notify generator thread g and assign d_i from D_x to solver thread s_n
[no assignable cube]
[else]
Cube solver thread s_n
$terminate_M$
$resume_M$
$suspend_M$
Add formula F_j
Interrupt solver thread s_n
$terminate_M$
Add all failed clauses in $\Psi(g)$, then clear it.
Unassign d_i from solver thread s_n
$suspend_M$
Solve cube c_i
$handleComm_M$
Filter
Store in $\Psi(s_n)$
Extract failed assumptions ϕ_i
[UNSAT]
[SAT]
Broadcast of the failed clauses
Store failed clause ψ_i in $\Psi(x)$
[else]
$[\phi_i = \emptyset]$
Mark job j as solved

Figure 6.5: Illustration of the solving loop of a cube solver thread s_n in the solving engine of a job node $p_x(j)$. Each smaller rectangle represents an action, each edge denotes a transition following the completion of the action. Dotted edges denote the start of concurrent actions or transitions to and from unfinished actions, in case of preemption. All of these cases are dependant on an asynchronous call of a method of the Mallob interface. Diamond shapes represent decisions. All actions in the exclusive block require the acquisition of an exclusive access right.

6.5 The distribution of failed clauses

In the following, we explain the distribution of failed clauses over the job tree. We do this using a graphical example given in the Figs. 6.6 and 6.7 and by explaining each possible behavior of a job node p_x on the left side.

(1) The aggregation of failed clauses is periodically initiated by the leaf nodes. Each leaf node p_x sends its locally accumulated set of failed clauses $\Psi(x)$ to its parent.

(2) If an inner node p_x has received a set of failed clauses from each direct child node, it merges them with its locally accumulated set $\Psi(x)$ and sends the result to its parent.

(3) The root node p_0 stores the merge result of the received sets and the locally stored failed clauses in $\Psi(0)$ to maintain a local record of all ever aggregated failed clauses. It then starts the broadcast procedure using $\Psi(0)$.

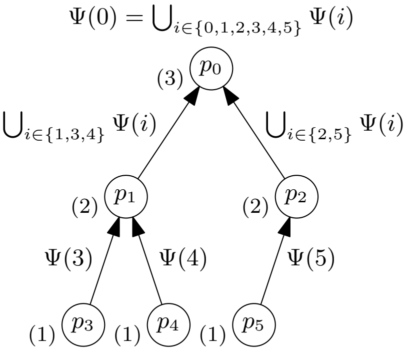

$\Psi(0) = \bigcup_{i\in\{0,1,2,3,4,5\}} \Psi(i)$
(3)
p_0
$\bigcup_{i\in\{1,3,4\}} \Psi(i)$
$\bigcup_{i\in\{2,5\}} \Psi(i)$
(2)
p_1
(2)
p_2
$\Psi(3)$
$\Psi(4)$
$\Psi(5)$
(1)
p_3
(1)
p_4
(1)
p_5

Figure 6.6: Aggregation of failed clauses.

The broadcast procedure consists of two actions on a set of failed clauses. The root node performs them on its local record. Every other node performs them on the received set of failed clauses of the broadcast.

- Filter the failed clauses for new ones and store these in $\Psi(s_n)$ of each local cube solver thread and in $\Psi(g)$ of the local generator thread g.
- Send the unfiltered set of failed clauses to all direct child nodes.

The broadcast then fades at the leaf nodes.

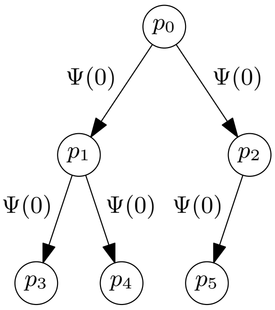


Figure 6.7: Broadcast of failed clauses.

6.6 The generator thread

Each job node $p_x(j)$ contains an additional thread for cube generation. The additional work reduces the CPU time of the threads that do the actual solving. However, it is expected to be negligible since only a few cubes are to be generated at a time. Similar to the solver threads, the generator thread is created and started on the call to $start_M$. It has the following two tasks:

1. Extend local cubes to generate new and more specialized cubes that can then be processed by the local cube solver threads or used to fulfill received requests. This is

done as long as there are cubes in D_x and until there are more than twice as many cubes in D_x than there are local cube solver threads.

2. If there are no cubes in D_x, the local control thread must be informed to send a request q_x for new cubes.

In the following we will go over the loop of the generator thread g of job node $p_x(j)$ which is depicted in Fig. 6.8. The generator thread g is based on a local instance of an IPASIR solver that contains the lookahead procedure $lookahead_C$ in its interface. So its first step is also to add the formula F_j to this instance. Then exclusive access to D_x is requested. This is required to assign a dynamic cube d_i from D_x to g. If D_x is empty and there is no pending request q_x, the wrapping solving engine is informed to request cubes. Then g sleeps until it is notified. If there are enough cubes in D_x or $\text{size}(D_x) = 0$ and there is a pending request q_x, g is directly put to sleep. A notification to the sleeping generator thread is caused by one of the following four reasons:

1. A local cube solver thread tries to assign a dynamic cube to itself.
2. The controlling thread of $p_x(j)$ attempts to fulfill a received request for cubes.
3. The request for cubes q_x was fulfilled and new cubes were received.
4. $suspend_M$ or $terminate_M$ was called.

After receiving a notification, g resumes to query D_x for an assignable cube. Eventually a dynamic cube d_i is assigned to g. This is followed by the addition of all newly received failed clauses to the local solver instance from $\Psi(g)$. Then g searches for a splitting literal l_{i+1} for the assigned cube c_i of d_i. When l_{i+1} is found, exclusive access to D_x is requested which is then used to check whether $d_i \in D_x$. If this is the case, replace the old cube with the two new ones and adjust a potential assignment to a cube solver thread. Then, or if $d_i \notin D_x$, repeat the loop by attempting to assign a new cube to g.

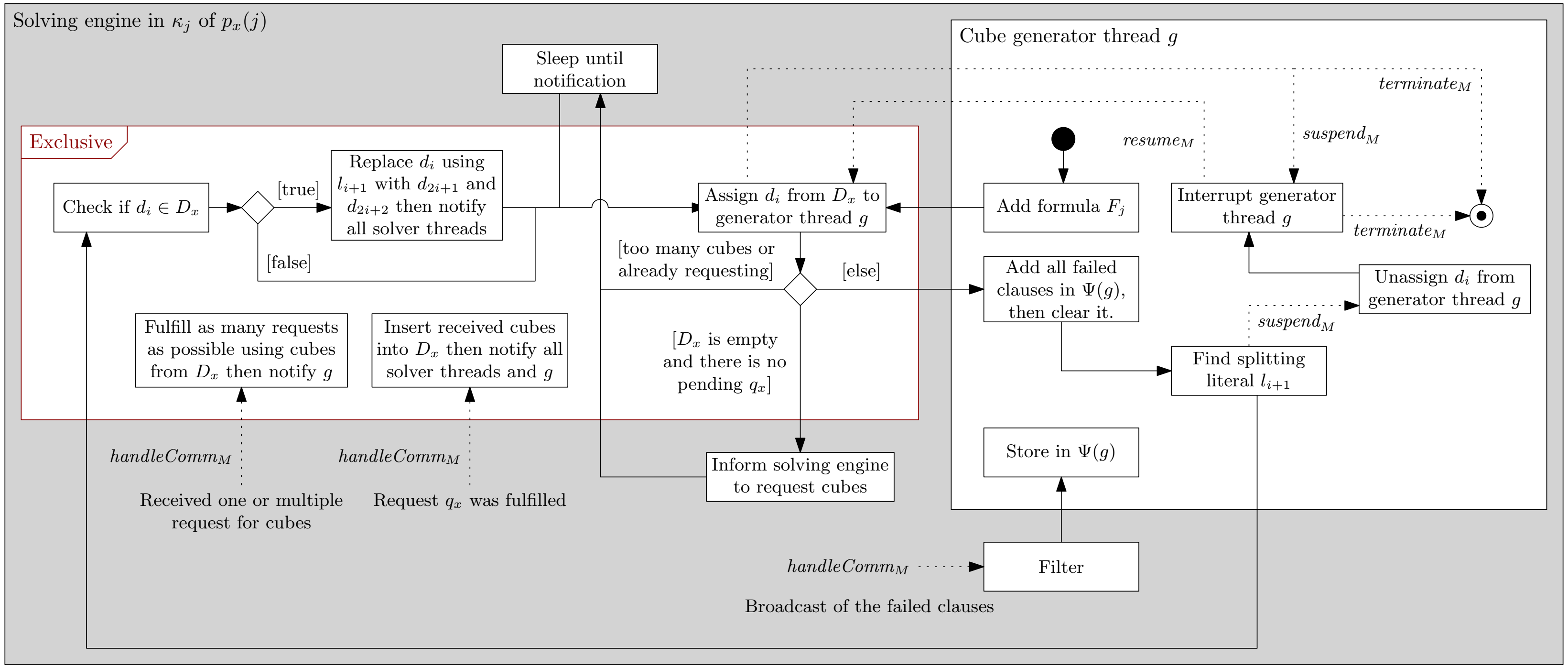

Solving engine in κ_j of $p_x(j)$
Cube generator thread g
Sleep until notification
Exclusive
Check if $d_i \in D_x$
[true]
[false]
Replace d_i using l_{i+1} with d_{2i+1} and d_{2i+2} then notify all solver threads
Assign d_i from D_x to generator thread g
[too many cubes or already requesting]
[else]
[D_x is empty and there is no pending q_x]
Fulfill as many requests as possible using cubes from D_x then notify g
Insert received cubes into D_x then notify all solver threads and g
$handleComm_M$
$handleComm_M$
Received one or multiple request for cubes
Request q_x was fulfilled
Inform solving engine to request cubes
$resume_M$
$suspend_M$
$terminate_M$
$terminate_M$
Add formula F_j
Interrupt generator thread g
Add all failed clauses in $\Psi(g)$, then clear it.
Unassign d_i from generator thread g
$suspend_M$
Find splitting literal l_{i+1}
Store in $\Psi(g)$
$handleComm_M$
Filter
Broadcast of the failed clauses

Figure 6.8: Illustration of the workflow of the cube generation thread g in the solving engine of a job node $p_x(j)$. Each smaller rectangle represents an action, each edge denotes a transition following the completion of the action. Dotted edges denote the start of concurrent actions or transitions to and from unfinished actions, in case of preemption. All of these cases are dependant on an asynchronous call of a method of the Mallob interface. Diamond shapes represent decisions. All actions in the exclusive block require the acquisition of an exclusive access right.

6.7 Handling of requests for cubes

In the following, we will explain the flow and handling of all types of cube requests over the job tree. We do this using a graphical example given in the Figs. 6.9 and 6.10 and by explaining each possible behavior of a job node p_x on the left side.

(1) The aggregation of cube requests is periodically initiated by the leaf nodes. If a leaf node p_x does not contain cubes and has no pending request, it sends q_x to its parent. Otherwise, it sends an empty set.

(2) If an inner node p_x has received a set of requests from each direct child node, it tries to fulfill as many as possible without exhausting D_x. Any unfulfilled requests are passed to its parent node. If p_x does not contain any cubes and has no pending request, q_x is also passed.

(3) The root node p_0 never passes requests that it could not fulfill. Instead, it stores them and tries to fulfill them at the end of the next aggregation.

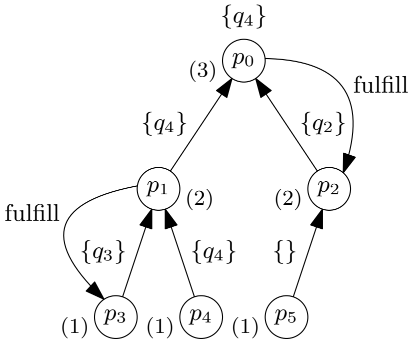


Figure 6.9: Aggregation of q_x with $x > 0$.

(4) If the root node p_0 does not contain any cubes, it sends the request q_0 to both direct child nodes.

(5) Is a receiving node p_x unable to fulfill the the request of the root, it passes q_0 to its direct children. If p_x is a leaf, q_0 is discarded.

(6) Is a receiving node p_x able to fulfill the request of the root, it does so by sending local free cubes from D_x to the root.

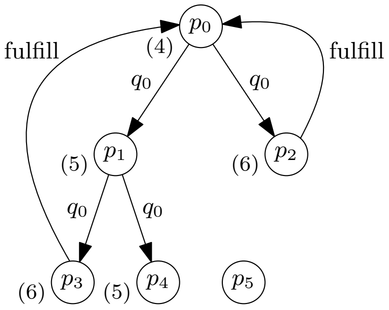


Figure 6.10: Broadcast of q_0.

The root nodes repeats the broadcast periodically until its q_0 has been fulfilled by at least one job node.

6.8 Malleability

The job node $p_x(j), x > 0$ may at any time be forced to pause its work on job j by a call to *suspend*$_M$. This is then achieved by all solver threads and the generator thread exiting their loop. This is possible if they are currently trying to assign a cube from D_x to themselves or are in the process of solving a cube or searching for a splitting literal respectively. In the second case, the solving or searching process is interrupted using $terminate_I$ and the assigned cube is unassigned. If a thread is currently performing a different action, its loop continues until one of the two mentioned actions is reached. All sleeping threads are notified on a call to $suspend_M$. After every thread of $p_x(j)$ has exited its solving loop, all local cubes in D_x and all newly accumulated failed clauses in $\Psi(x)$ are sent to the root node $p_0(j)$. This is done to guarantee that all existing cubes are always contained by active job nodes. This also requires the termination of each solver thread to happen synchronously to the control thread's call to $suspend_M$. The local cubes and clauses are then sent as a parting message at the end of the method. This suspension procedure forces a resumed job node $p_x(j)$ to request new cubes and to restart each contained thread.

When $terminate_I$ is called, each working thread exits its loop in the same way as on a call to $suspend_M$. However, since the job has reached the end of its life cycle no cubes or clauses are sent.

6.9 Handling of delayed messages

A PE can leave the job it is currently working on at any time. Therefore, it must be able to correctly handle delayed messages from a previous job j even if it has already finished its work j. Delayed messages can be received if they were sent shortly before the PE left the job. Their correct handling is necessary to prevent the loss of sent failed clauses or cubes.

If a PE r receives aggregated failed clauses of a suspended job j, they are sent to the root node $p_0(j)$. This is necessary to not lose the solving result of a now pruned cube. Any received broadcasts are ignored. This is correct even if r later continues work on j, since a subsequent broadcast will contain the missed failed clauses.

If r receives the fulfillment of a previously sent request for cubes of a now suspended job j, it sends the received cubes to the root node $p_0(j)$. This guarantees that the cubes are not lost. Every received requests is discarded. This is possible for aggregated requests, since all former child nodes must now also be suspended. Root requests can always be discarded because they are not unique.

7 Evaluation

In this chapter we do an evaluation of our two approaches. This is done by passing a set of SAT problems to the Mallob framework which then malleable schedules the work on them using the specified approach. Note that in this mode one PE takes over the role of the client and therefore does not participate in the solving algorithm. In addition, the randomized scheduling and load balancing paradigm of Mallob requires to keep at least one PE idle at all times to ensure quick adaptation of a new job request. Further, we control the degree of parallelization on each formula by limiting the number of active jobs J. All other options of Mallob that affect the load balancing and scheduling are kept at their default value.

For the evaluation, the benchmark set from the SAT Competition 2020 is used which consists of 400 problems. However, we removed the formula

```
sv-comp19_prop-reachafety.queue_longer_false-unreach-call.i-witness.cnf
```

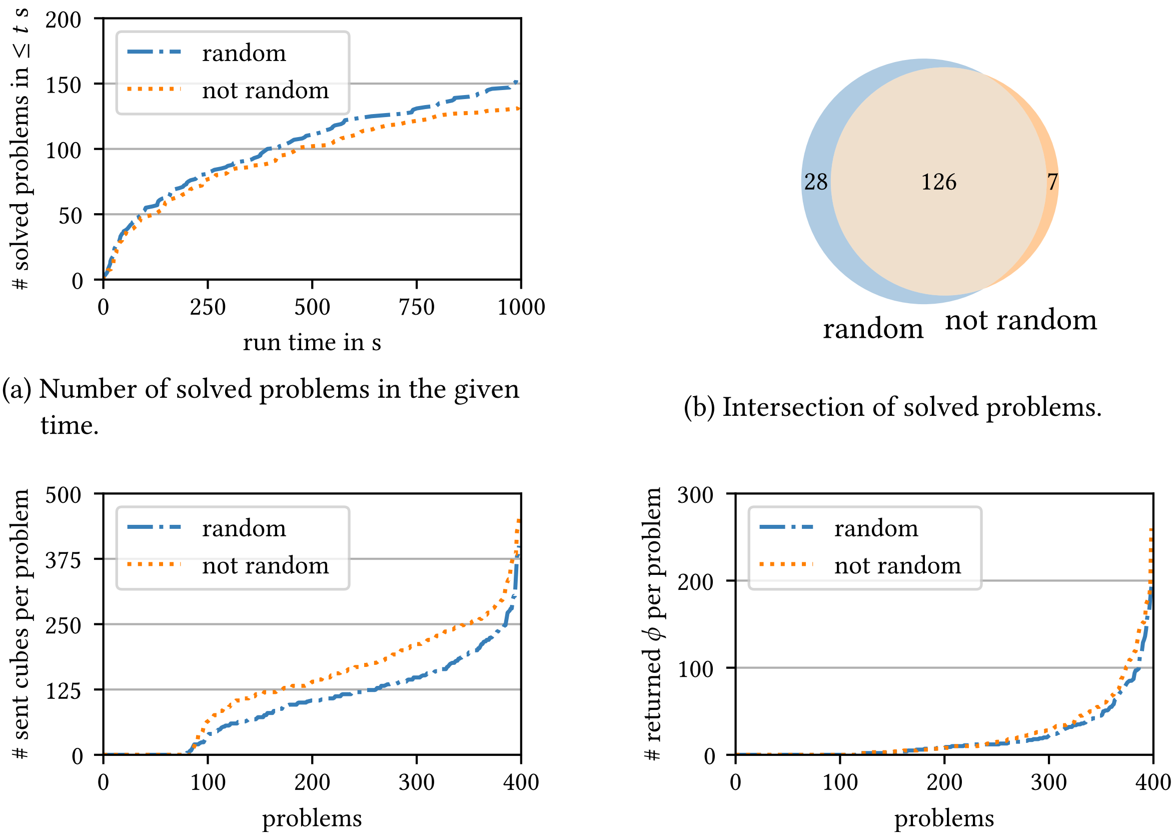
because the solver working on it became unresponsive by not allowing termination. The time limit per problem is set to 1000 seconds. This corresponds to the configuration of the Cloud Track of the SAT Competition 2020. In the following, we exclusively use the state-of-the-art SAT solver Cadical for cube generation and solving.

7.1 Hardware

All benchmarks were done on a machine that uses an AMD EPYC™ 7702P with 1024GB main memory. This processor features 64 real cores that support 128 threads via hyper-threading. Each core uses a base clock of 2.0GHz that can boost up to 3.35GHz. Each thread has its own level one cache, each core has a personal level two cache and 4 cores share a level three cache. The cache layout is important because we use it later to partition the processor into several identical virtual processing elements to simulate a distributed computing environment for Mallob.

7.2 Static Cube&Conquer

In the following, our first approach is evaluated. We begin by comparing some of the parameters used and conclude with an evaluation of its scalability. We do not evaluate any run-time characteristics since we assume that each worker always receives work due to the used work distribution strategy.



(a) Number of solved problems in the given time.

(b) Intersection of solved problems.

(c) Number of sent cubes per problem in ascending order.

(d) Number of returned sets of failed assumptions per problem in ascending order.

Figure 7.1: Performance of Static Cube&Conquer using the configuration from section 7.2.1 with randomization activated and disabled.

7.2.1 Randomization

We start evaluating the first approach by testing the randomization assumption from Section 5.3.3. This is done using the following configuration: Each core represents a single PE. Each worker therefore only contains a single solver thread. If we subtract the client and the idle PE, we get 62 active workers. They work on $J = 4$ active jobs at a time. We can therefore expect each job to eventually contain 15-16 workers. The height h of the cube tree is set to 7. This forces the manager to generate $2^7 = 128$ cubes. These are distributed to the workers in batches of maximum 4 cubes (*cubes-per-worker* = 4).

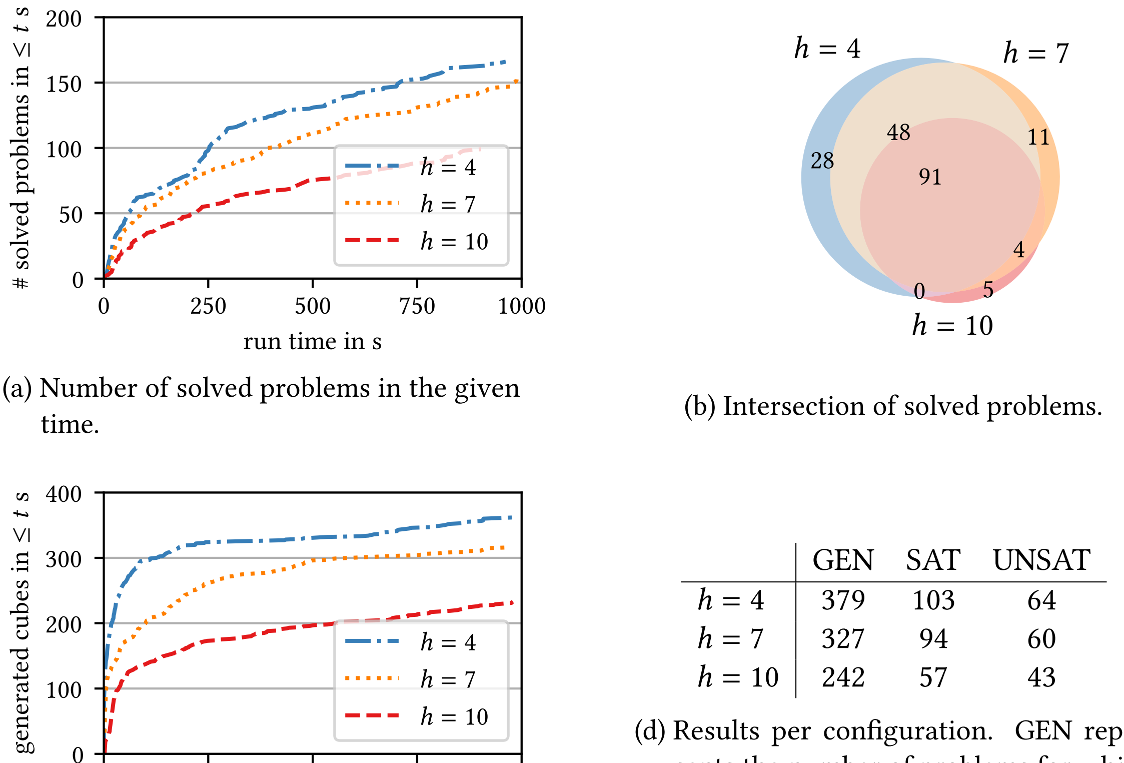
In the Figs. 7.1a and 7.1b we see that the random configuration solves more problems in the same time. One possible reason for this is that each solver works on more relevant cubes, as Fig. 7.1c shows less cubes being sent per problem, and in Fig. 7.1d we see that less failed assumptions are returned. The problems where no cubes were sent were interrupted by a timeout during the cube generation step. Since randomization performs better, we will always randomize in the following benchmarks.

7.2.2 Height of the cube tree

As the next step, we evaluate the effect of different cube tree heights. A higher cube tree should result in more and easier cubes, but increases the run time of the first phase of the algorithm. We will do this using the following three configurations:

1. $h = 4 \longrightarrow 16$ cubes, *cubes-per-worker* = 1
2. $h = 7 \longrightarrow 128$ cubes, *cubes-per-worker* = 4
3. $h = 10 \longrightarrow 1024$ cubes, *cubes-per-worker* = 64

Everything else is set identical to the configuration from Section 5.3.3 with enabled randomization.



(a) Number of solved problems in the given time.

(b) Intersection of solved problems.

(c) Number of problems whose cubes were generated in the given time.

	GEN	SAT	UNSAT
$h = 4$	379	103	64
$h = 7$	327	94	60
$h = 10$	242	57	43

(d) Results per configuration. GEN represents the number of problems for which the cube generation process was finished.

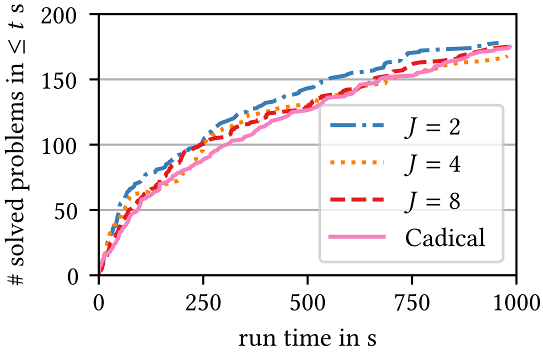
Figure 7.2: Performance of Static Cube&Conquer using different cube tree heights and batch sizes.

In Fig. 7.2a we see that the extra effort required to generate more cubes is generally not useful in a time-limited scenario. The Fig. 7.2c and the Table in 7.2d show that a larger cube tree drastically increases the duration of the first step of the algorithm. This leads to a reduction of the remaining solving time and for numerous of problems to not even reach the cube solving phase. This results in the remaining solving time being shortened and many problems not even reaching the cube solving phase. Fig. 7.2b shows that forming simpler cubes only leads to the solving of 5 + 4 + 11 = 20 additional problems that were not solvable with a tree height of 4.

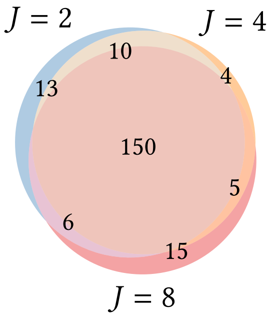
7.2.3 Scaling

The last evaluation of the first approach addresses its scalability. We do this by varying the limit of active jobs in the system J while using the best configuration from Section 7.2.2. However, we will adjust h to J to avoid generating fewer cubes than workers per job in a balanced scenario. We chose the following three configurations:

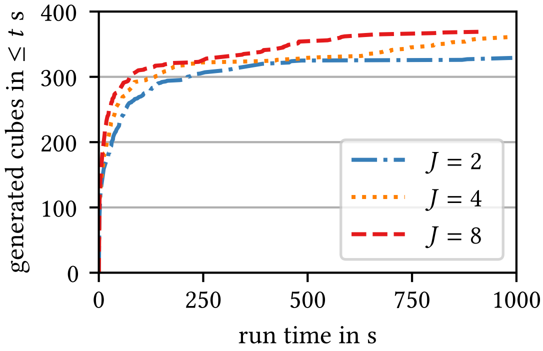
1. $J = 2 \longrightarrow$ 31 workers per job, $h = 5 \longrightarrow$ 32 cubes
2. $J = 4 \longrightarrow$ 15-16 workers per job, $h = 4 \longrightarrow$ 16 cubes
3. $J = 8 \longrightarrow$ 7-8 workers per job, $h = 3 \longrightarrow$ 8 cubes



(a) Number of solved problems in the given time.



(b) Intersection of solved problems.



(c) Number of problems whose cubes were generated in the given time.

	Total		Extra	
	SAT	UNSAT	SAT	UNSAT
$J = 2$	114	65	40	4
$J = 4$	104	65	31	3
$J = 8$	106	70	28	4

(d) Results per configuration. The two right-most columns contain the problems solved by the configuration, but not by Cadical.

Figure 7.3: Performance of Static Cube&Conquer using different active jobs limits.

In Fig. 7.3a we see that $J = 2$ outperforms Cadical. The next best configuration is $J = 8$. $J = 4$ is occasionally surpassed by Cadical. So we can say that the approach is not guaranteed to scale by adding workers and increasing the cube tree accordingly. One possible reason for this is that the simplicity of the cubes does not scale linearly with their size. It is therefore not guaranteed that the resulting solving speedup outweighs the extra work required to generate the additional cubes. The required extra work time to generate the cubes is shown in Fig. 7.3c. Working on different cubes also changes the set of solved problems. This is shown in Fig. 7.3b.

However, overall we can say that we found a configuration in which our first approach outperforms the internally used SAT solver Cadical. In Fig. 7.3d we compare the solved problems of our three configurations and Cadical. It is noticeable that our approach works better on problems that are satisfiable. This can be explained by its termination scenarios.

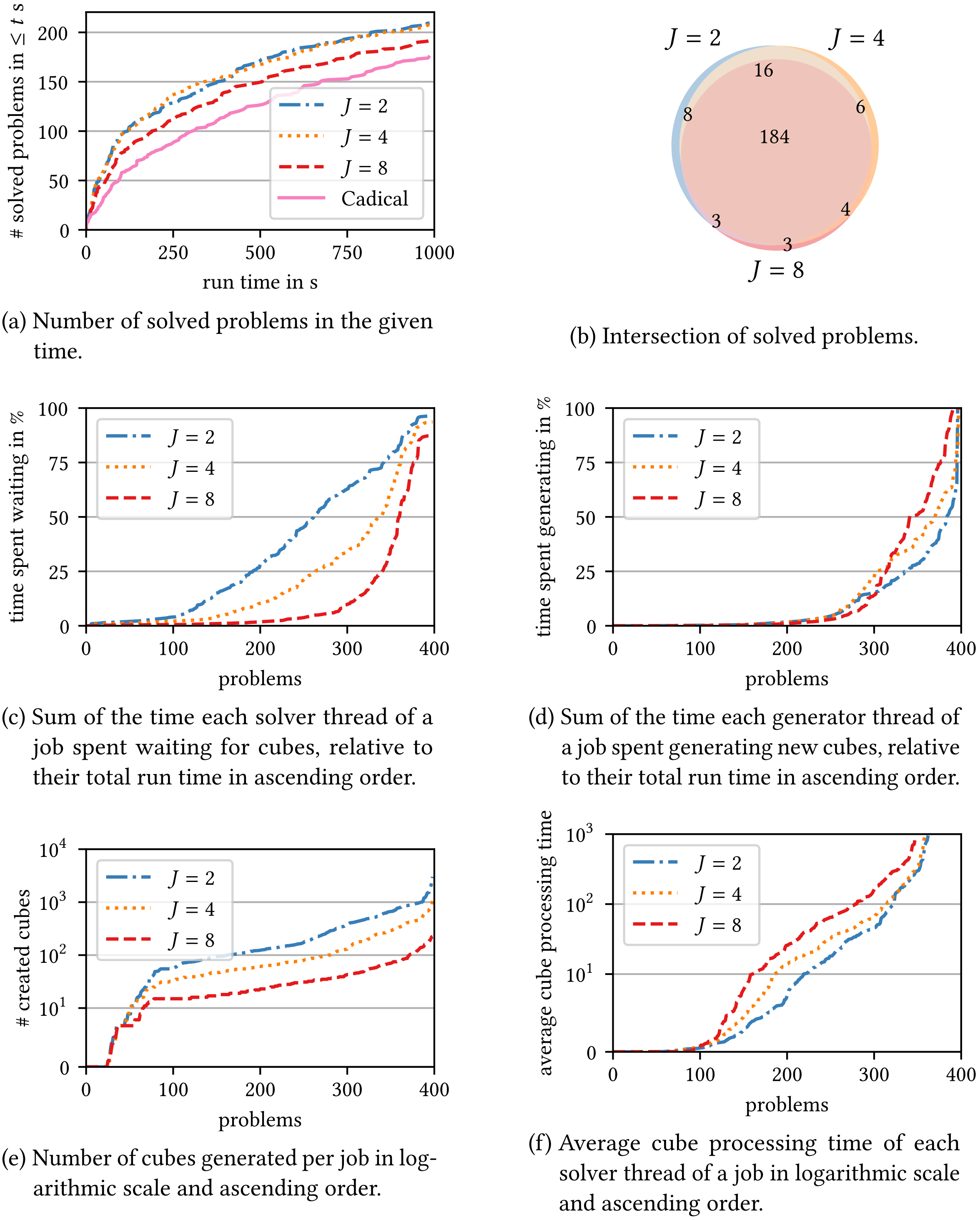
7.3 Dynamic Cube&Conquer

In this section we evaluate our second approach. As the approach is not based on parameters, we directly focus on its scalability. To increase the number of solver threads relative to the generator threads, we define each PE to consist of four cores of the CPU. We bundle the cores according to their level three cache. Each job node therefore possesses four cube solver threads and a single cube generation thread. Subtracting the client and the idle PE, we expect 14 simultaneously active PEs. We chose the following three configurations:

1. $J = 2 \rightarrow$ 7 job nodes per job
2. $J = 4 \rightarrow$ 3-4 job nodes per job
3. $J = 8 \rightarrow$ 1-2 job nodes per job

In Fig. 7.4a we see that all three configurations outperform Cadical. Also the approach scales well from $J = 8$ to $J = 4$. From $J = 4$ to $J = 2$ however, the number of solved problems only increases by a single instance and the run times stay at a similar level. This can be explained by Fig. 7.4c. It shows that for $J = 2$ the wait time of each solver thread drastically increases. Also, Fig. 7.4d shows that the generator threads of $J = 2$ are the most often idle. Probably because they also lack cubes to create new cubes. One could therefore argue that not enough cubes are generated. However, in Fig. 7.4e we see that for $J = 2$ many more cubes are generated than for $J = 4$. Another possible cause for the bad scaling could therefore be that the generator threads produce ineffective cubes that fail to split the problem into relevant sub-problems. This is further supported by Fig. 7.4f, which shows that the average cube processing time of the solver threads decreases sharply from $J = 4$ to $J = 2$.

In total this approach outperforms the first one from Section 7.2.3 for any number of active cubes. Comparing the venn diagram in Fig 7.4b with the one in Fig. 7.3b, we can see that this approach has a lower variance of solved problems per configuration.



(a) Number of solved problems in the given time.

(b) Intersection of solved problems.

(c) Sum of the time each solver thread of a job spent waiting for cubes, relative to their total run time in ascending order.

(d) Sum of the time each generator thread of a job spent generating new cubes, relative to their total run time in ascending order.

(e) Number of cubes generated per job in logarithmic scale and ascending order.

(f) Average cube processing time of each solver thread of a job in logarithmic scale and ascending order.

Figure 7.4: Performance of Dynamic Cube&Conquer using different active job limits.

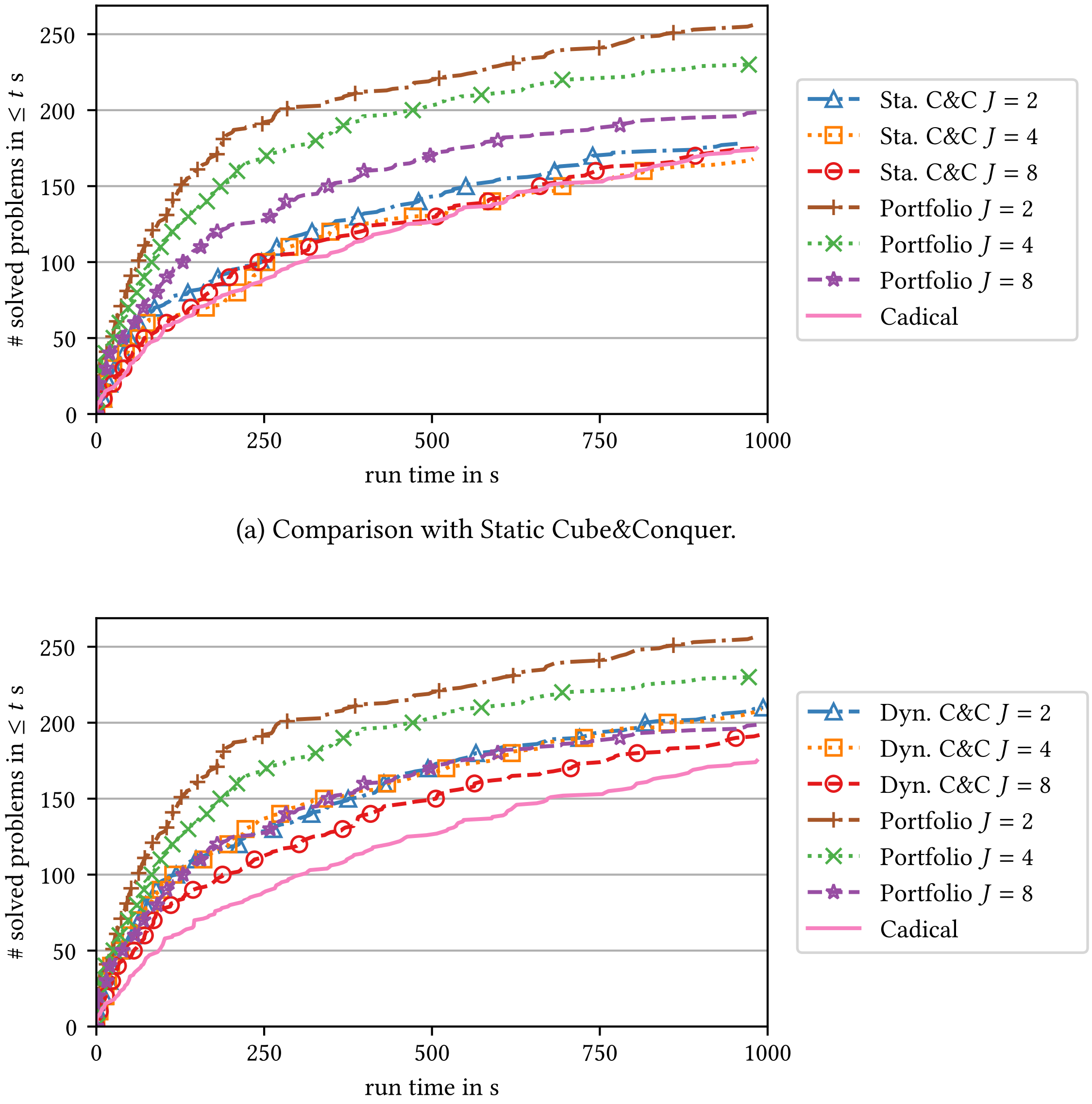
						Total		Extra	
	solved	on c_0	on $c_i, i > 0$	$\Psi = \emptyset$	$\Psi \neq \emptyset$	SAT	UNSAT	SAT	UNSAT
$J = 2$	211	129	82	142	69	129	82	39	7
$J = 4$	210	123	87	157	53	129	81	39	6
$J = 8$	194	98	96	157	37	116	78	27	4

Table 7.1: Number of solved problems per configuration. The number is further divided into the number of problems that were solved using the root cube c_0 or on a sub-formula and if the solver did add any failed clauses ψ. The solved problems are also divided into the two possible results and into the problems that were not solved by Cadical on its own.

In the Table 7.1 we see the success of this approach. About two fifths to half of the problems were solved using a generated cube. For these problems, the approach is better suited than an unmodified Cadical. Also, a non-negligible amount was solved with the help of added failed clauses. In comparison with the table in Fig. 7.3d, we see that the configurations $J = 2$ and $J = 4$ of this approach both solve more extra problems than the best one of our first approach. However, since the difference is marginal, we can conclude that this approach performs better because it also solves the problems that were previously only solved by standalone Cadical. It therefore combines the advantages of Cube&Conquer and the state-of-the-art SAT solver Cadical. It also scales much better for $J = 4$.

7.4 Comparison with the portfolio-based solving engine

Finally, we compare the three best configurations of both approaches with the malleable SAT solving engine based on HordeSat included in Mallob. To ensure comparability we also add the performance of standalone Cadical. In Fig. 7.5a we compare our first approach. It shows that all configurations are outperformed by the better scaling portfolio-based solution. In Fig. 7.5b we see that Dynamic Cube&Conquer with $J = 2$ and $J = 4$ outperforms the portfolio-based solution with $J = 8$. The other two configuration are not surpassed because they scale much better.



(a) Comparison with Static Cube&Conquer.

(b) Comparison with Dynamic Cube&Conquer.

Figure 7.5: Comparison with the portfolio-based solution in Mallob.

8 Conclusion and Future Work

In this chapter, we give a conclusion of our accomplishments and propose possible future work.

8.1 Conclusion

To expand the field of malleable SAT solving, we specified two strongly differing parallel SAT solving methods based on the search-space splitting approach that allow for malleability. Both were implemented using Mallob to further diversify its SAT solving capabilities. The methods are designed around a solver that implements the IPASIR interface. This allows them to be used with various sequential solvers.

Classic Cube&Conquer requires an initial step to generate a predefined set of cubes. This is done in our static approach after problem adoption in a sequential manner. The demand-based scheduling is used to ensure that only a single processing element is occupied at this stage. We then incorporate malleability into the highly parallel cube solving process by not strictly binding the generated cubes to solvers. Instead, we present a cube distribution strategy that distributes each unsolved cube evenly and repeatedly until it is solved. This allows each solver to leave the method at any time without affecting its completeness. We tested this method to work best if the cubes are distributed in a random order. Also, we concluded that its main limiting factor in a time-critical solving scenario is the additional step that is required to generate the cubes. This hinders scaling, since providing a workload that can be balanced across a large number of solvers requires an even larger number of cubes. However, we found configurations that allow this method to outperform the internally used SAT solver in standalone mode. Using these, we noticed that the method works well in solving additional problems that are satisfiable.

Allowing for malleability in our dynamic method based on distributed and continuous cube generation was significantly more difficult. Since we decided against a designated manager to avoid a communication bottleneck, each assigned processing element must carefully manage its held cubes and solving results. This is achieved through interleaved invariants and specialized communication procedures. Edge cases due to preemption are handled via fallback strategies. To enable collaborative SAT solving, we introduce a new approach based on sharing clauses from failed assumption literals. It allows passing insights about solved sub-problems. We also use it to improve the quality of the generated cubes. In the evaluation, we have shown that the dynamic approach significantly outperforms the internally used solver and the first approach even with a low degree of parallelization. It therefore makes better use of the given resources. However, it did not scale well for a larger number of processing elements in our test. By looking at several run-time characteristics, we conclude that this is caused by a sharp decrease in the quality of the generated cubes. Because of this, the approach is outperformed by the well-scaling portfolio-based SAT solving method included in Mallob.

8.2 Future work

In this section, we propose future work that builds on the approaches of this work.

Improve the cube generation step of Static Cube&Conquer In the evaluation of our first approach, we saw that its main limiting factor is the sequential initial step. In a timed scenario, only a few cubes may be created. Otherwise, the actual parallel SAT solving time is immensely reduced. This is a major disadvantage because the number of cubes determines the maximum achievable degree of parallelization. A possible solution would be to expand this step to also make use of the massively parallel distributed environment. Another idea would be to interleave both steps and to set the demand according to the already generated workload. Falling back to an easier heuristics is also a possible solution.

Improve the cube quality of Dynamic Cube&Conquer The bad scaling of our second approach is most likely caused by the shortage of good cubes that split the problem into reasonably sized sub-problems. There are several ideas how to approach this problem: By increasing the number of cube generators, more cubes will be created. Heuristically, this should increase the number of found good cubes. Another idea is to support the cube generators with learned clauses from the solver threads. This should allow them to simplify the formula and thus increase the quality of their generated cubes. We could also introduce lookback heuristics by evaluating the state of a solver that has been working on a cube for a long time and use it to generate cubes.

Periodically test the cubes in Dynamic Cube&Conquer A current imperfection of our second approach is that a solver continues to work on a cube c even if the formula was proven to be unsatisfiable for a subset of the assumptions in c. This could be addressed by periodically testing whether this is the case and, if so, interrupting the solver. This would also add a new termination scenario when the solver on the root cube is interrupted.

Bibliography

[1] Gilles Audemard et al. "An adaptive parallel SAT solver". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 30–48.

[2] Gilles Audemard et al. "An effective distributed D&C approach for the satisfiability problem". In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE. 2014, pp. 183–187.

[3] Tomáš Balyo, Peter Sanders, and Carsten Sinz. "HordeSat: A massively parallel portfolio SAT solver". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2015, pp. 156–172.

[4] Tomáš Balyo et al. "Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions". In: (2020).

[5] Tomáš Balyo et al. "SAT race 2015". In: *Artificial Intelligence* 241 (2016), pp. 45–65.

[6] Armin Biere. "Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018". In: *Proceedings of SAT Competition* (2017), pp. 14–15.

[7] Armin Biere, Marijn J. H. Heule, and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[8] Koen Claessen et al. "SAT-solving in practice". In: *2008 9th International Workshop on Discrete Event Systems*. IEEE. 2008, pp. 61–67.

[9] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. 1971, pp. 151–158.

[10] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[11] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[12] Travis Desell, Kaoutar El Maghraoui, and Carlos A Varela. "Malleable applications for scalable high performance computing". In: *Cluster Computing* 10.3 (2007), pp. 323–337.

[13] Armin Biere Katalin Fazekas Mathias Fleury and Maximilian Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020". In: *SAT COMPETITION 2020* (2020), p. 50.

[14] Richard L Graham et al. "Open MPI: A high-performance, heterogeneous MPI". In: *2006 IEEE International Conference on Cluster Computing*. IEEE. 2006, pp. 1–9.

[15] Jun Gu et al. *Algorithms for the satisfiability (SAT) problem: A survey*. Tech. rep. Cincinnati Univ oh Dept of Electrical and Computer Engineering, 1996.

[16] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. "ManySAT: a parallel SAT solver". In: *Journal on Satisfiability, Boolean Modeling and Computation* 6.4 (2010), pp. 245–262.

[17] Maximilian Heisinger, Mathias Fleury, and Armin Biere. “Distributed Cube and Conquer with Paracooba”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2020, pp. 114–122.

[18] Marijn J. H. Heule et al. “Cube and conquer: Guiding CDCL SAT solvers by lookaheads”. In: *Haifa Verification Conference*. Springer. 2011, pp. 50–65.

[19] Marijn J. H. Heule et al. “March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2004, pp. 345–359.

[20] Matti Järvisalo et al. “The international SAT solver competitions”. In: *Ai Magazine* 33.1 (2012), pp. 89–92.

[21] Daniela Kaufmann et al. “Arithmetic verification problems submitted to the SAT Race 2019”. In: *Proc. of SAT Race* 2019 (2019).

[22] Ludovic Le Frioux et al. “Modular and efficient divide-and-conquer SAT solver on top of the painless framework”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2019, pp. 135–151.

[23] Joao Marques-Silva. “Practical applications of boolean satisfiability”. In: *2008 9th International Workshop on Discrete Event Systems*. IEEE. 2008, pp. 74–80.

[24] Ruben Martins, Vasco Manquinho, and Inês Lynce. “An overview of parallel SAT solving”. In: *Constraints* 17.3 (2012), pp. 304–347.

[25] Dominik Schreiber. “Engineering HordeSat Towards Malleability: mallob-mono in the SAT 2020 Cloud Track”. In: *Proceedings of SAT competition 2020*, p. 45.

[26] Dominik Schreiber and Peter Sanders. “Scalable SAT Solving in the Cloud”. In: *Proceedings of SAT competition 2021*. (under revision).

[27] *The International SAT Competition Web Page*. Feb. 2021. URL: `http://www.satcompetition.org/`.