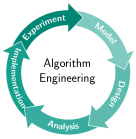


Sharing Clauses Across Different Problems in Distributed SAT Solving

Bachelor's Thesis of

Samuel Born

at the Department of Informatics
Institute for Theoretical Computer Science, Algorithm Engineering

Reviewer: Prof. Dr. rer. nat. Peter Sanders
Advisor: M.Sc. Dominik Schreiber

01. December 2022 – 31. March 2023

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. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

Karlsruhe, March 31, 2023

. . . . . .

(Samuel Born)

Abstract

Conflict-Driven Clause Learning (CDCL) is one of the most popular approaches for solving SAT problems. During the execution of the algorithm, the solver learns redundant clauses to speed up the further solving of the problem. In order to improve the performance, these clauses can also be shared in parallel SAT solvers between solver instances of the same problem.

In this thesis, we present a communication framework for Mallob, that enables groups of jobs to share information. We use this framework to share clauses not just between solver instances for the same problem, but also across different problems. Since the problems are in most cases not identical, we must verify the applicability of each shared clause in order to not learn wrong information. Our verification process is similar to the incomplete redundancy check of clause strengthening. For identical instances that only differ in assumptions, we can skip the validation step.

Our experimental results indicate that, for some instances, our validation approach performs marginally better when cross-problem clause sharing is utilized. However, we observed that skipping the validation step can result in a statistically significant speedup.

Zusammenfassung

Conflict-Driven Clause Learning (CDCL) ist einer der gängigsten Ansätze zum Lösen von SAT-Problemen. Während der Ausführung des Algorithmus lernt der Algorithmus redundante Klauseln, um das weitere Lösen des Problems zu beschleunigen. Um die Performance zu verbessern, können diese Klauseln auch in parallelen SAT-Solvern zwischen Solver-Instanzen desselben Problems geteilt werden.

In dieser Arbeit stellen wir ein Kommunikations-Framework für Mallob vor, das es Gruppen von Jobs ermöglicht, Nachrichten auszutauschen. Wir haben dieses Framework verwendet, um Klauseln nicht nur zwischen Solver-Instanzen für dasselbe Problem, sondern auch zwischen unterschiedlichen Problemen auszutauschen. Da die Probleme im Normalfall nicht identisch sind, müssen wir die Anwendbarkeit jeder gemeinsam genutzten Klausel verifizieren, um keine falschen Informationen zu erhalten. Dieser Validierungsschritt ähnelt dem incomplete redundacy check des clause strengthenings. Bei identischen Instanzen, die sich nur in den Annahmen unterscheiden, können wir den Validierungsschritt überspringen.

Unsere experimentellen Ergebnisse deuten darauf hin, dass unser Validierungsansatz für einige Instanzen marginal besser abschneidet, wenn das Teilen von Klauseln über mehrere Probleme hinweg verwendet wird. Es ist erwähnenswert, dass das Überspringen des Validierungsschritts zu einer statistisch signifikanten Beschleunigung führen kann.

Contents

1 Introduction

1.1 Motivation

Imagine you and your friends are working on different problems on a math worksheet. Though you each have different tasks to complete, you realize that some of the problems are similar and you may be able to share partial results with one another. However, it's important to be cautious, as a partial result from a friend may not be applicable to your problem. For example, in your task, you may be calculating over another field, and a partial result from a friend may be incorrect in your context. To ensure the validity of the shared information, you first verify its applicability before incorporating it. However, since you already know the solution of the partial problem, it is easier for you to check if it is applicable. By doing so, you and your friends can finish the worksheet more quickly and head to the cafeteria sooner.

In this thesis, we apply the same strategy to SAT-problems. The SAT problem is the problem of determining if there exists a variable assignment that satisfies a given boolean formula. A popular approach for the sequential solving of SAT problems is Conflict Driven Clause Learning. CDCL solvers do an intelligent depth-first search over all variable assignments. If these solvers encounter a contradiction, it is recorded in a redundant clause. These clauses can be used to make the further search more efficient and find new contradictions faster.

In recent years, parallel computing has become more and more important to build fast software. State-of-the-art SAT solvers also take advantage of parallel computing. A popular approach often used by parallel SAT solvers is portfolio solving. Here, different and differently parameterized solvers solve the problem simultaneously in the hope that one of them is well suited for the problem at hand. Each solver can pass partial results (clauses) on to other solver instances for the same problem.

We will extend this, by not only sharing clauses across solver instances of the same problem, but also share clauses across *different*, but similar, problems while also checking their validity.

1.2 Contribution

We build on the success of clause sharing for individual problems so that the solving procedure can be applied to different but similar problems. We identify and develop the two major steps for enabling cross-problem clause sharing. Our first contribution, is a general communication framework for the communication across problems. Secondly, we use this framework to share clauses across problems and afterward validate them with our clause validation scheme. Additionally, we discuss instances where it is possible to

tun off this validation scheme. On instances where we need our clause validation scheme, we saw a slight speedup for some of our tested instances. On instances where we could disable clause validation, we saw a statistically significant speedup of around 10% for most instances.

For the implementation, we work on the existing software stack of Mallob. Mallob is a platform for massively parallel and distributed on-demand processing of malleable jobs, handling their scheduling and load balancing [17]. Mallob can also be used to solve multiple SAT problems simultaneously in an orchestrated manner, and therefore already provides many functions for communication between jobs. This makes it easy to adapt the software to our needs.

1.3 Structure of Thesis

Chapter 2 gives a quick overview over SAT Solving and techniques we will use in our approach. We will discuss the concept of clause strengthening and highlight the similarities with our approach. Afterward, we explain our approach for solving different but similar SAT problems with clause sharing and evaluate its performance. At last, we conclude our results in Chapter 5 and point out possibilities for future work.

2 Preliminaries

In this chapter we will first clarify some fundamentals that we will need in our approach and afterward introduce some related work.

2.1 SAT Solving

The Boolean satisfiability problem (SAT) is the problem of determining if there exists a truth value assignment that satisfies a given boolean formula F. F is typically given in Conjunctive normal form (CNF). A formula is in CNF if it is a conjunction of one or more clauses, e.g. $C_1 \wedge C_2 \wedge C_3$. A clause is a disjunction of one or more literals, e.g. $L_1 \vee L_2 \vee L_3$. A literal is a variable V or a negated variable $\neg V$.

Given such a formula F, now we have to find out if there is a variable assignment with `true` and `false` so that the formula evaluates to `true` . If there is such a variable assignment, we call the formula satisfiable. If the formula evaluates to `false` with all variable assignments, we call the formula unsatisfiable.

The SAT problem is NP-complete. This means that no algorithm is known that can solve the problem in polynomial time. Due to the general problem definition, many other NP-hard problems can be easily reduced to the SAT problem. Thus, if you have a fast solver for SAT problems, you may have a fast solver for many other problems.

Most modern solvers imply an approach to solving SAT Problems called Conflict-driven Clause Learning. To better explain these, we will first introduce the concept of unit propagation.

Unit Propagation Given a literal L, we define $F|_l$ to be the CNF formula resulting in removing all clauses with l (the clause is satisfied as L would satisfy it) and all occurrences of $\neg L$ ($\neg L$ can not satisfy the clause anymore, as it would already be assigned the opposite truth value). We can recursively define Unit Propagation (UP) in the following way [14]:

$$UP(F) = \begin{cases} F, & \text{if F does not contain any unit clause.} \\ \bot, & \text{if F contains two unit clauses } \{V\} \text{ and } \{\neg V\}. \\ UP(F|_l), & \text{otherwise} \end{cases}$$

Conflict-driven Clause Learning A Conflict-driven Clause Learning (CDCL) solver [13] performs a non-chronological backtrack search in the space of partial truth assignments [12]. Concretely, the solver repeatedly picks a decision literal L_i and applies unit propagation $UP(F \cup \{L_1, \ldots, L_i\})$ until the empty formula or the empty clause is derived [12]. $\{L_1, \ldots, L_{i-1}\}$ are decision variables from earlier iterations. If the empty formula is derived, the problem is satisfiable, as all clauses are satisfied. If the empty clause is derived, we

found a contradiction. The truth assignment, that the algorithm did choose, was incorrect. In this case, the reasons are analyzed, and a new clause is learned; normally via the First Unique Implication Point Scheme [19]. How exactly this happens is not of further relevance to this work. This newly learned clause can help to find contradictions of new decision literals faster. Inspired by [1], we illustrated the pseudocode of CDCL solvers in Algorithm 1.

Learning too many clauses though can also slow down the solver and may overflow the available memory [12]. This is why most solvers periodically delete a subset of the learned clauses.

Algorithm 1 Pseudocode of CDCL solvers

```
Input: CNF formula F
  while not all variables assigned do
      assignDecisionLiteral
      doUnitPorpagation
      if conflict detected then
          analyzeConflict
          addLearnedClause
          backtrack or return UNSAT
      end if
  end while
  return SAT
```

Parallel SAT solving There are two major approaches for parallelizing SAT algorithms that can be distinguished [18].

The classical divide-and-conquer approach to parallelizing SAT solving is to split the search space between the search engines such that no overlap is possible. This is usually done by starting each solver with a different fixed partial assignment [1].

The second, so-called *portfolio* approach, is to run several solvers with different configurations in parallel. The solvers can be completely different solvers or the same solver with different parameters. The system finishes whenever the fastest solver is done. The hope is that one of the solvers might fit the problem well, and finishes fast. When there is no interaction between the solvers of a portfolio, we more precisely call it a *pure portfolio* [1]. The first solver to implement this kind of approach was ManySAT [7]. Another example for an efficient and modern portfolio solver is also featured in our framework Mallob.

To boost the performance of both of these approaches, one can incorporate Clause sharing, to exchange important partial results with other solvers. In the case of portfolio solvers, such a portfolio is called *cooperative portfolio* [18]. ManySAT also implemented this [7]. Because we are dealing with clauses being shared across different problems in this work, we will call those kind of shared clauses within a portfolio solver *internally shared clauses*.

2.2 Mallob Nomenclature

Mallob is the framework we will be using to implement our approach. Mallob already features an engine for distributed SAT solving. To have clear and precise wording in this thesis, we will now clarify the nomenclature of mallob.

Given a boolean formula F. The goal of checking this formula is encapsulated in mallob by a so-called *job*. Different numbers of *workers* are assigned to these jobs depending on priority and demand. Each worker corresponds to a set of computing resources and can actively contribute to solving the problem. A job can have multiple workers assigned to it, but it always has at least one. This one worker is the root worker of the job, and we will call him *job representative*.

When dealing with cross problem clause sharing, similar jobs are gathered into a ring structure. This ring structure has a *group representative*. The group representative handles join requests of other job representatives.

2.3 Approximate Membership Query

Approximate Membership Query Data Structures (AMQ) are space-efficient probabilistic data structures that enable constant-time membership queries [3]. When queried, an AMQ returns either if that element is not - or if it might already be - in the set. False positives may occur, but false negatives cannot. While it is possible to add elements to the set, in most cases removal is not supported. As the number of items in the set increases, typically the probability of false positives rises.

An example for a very simple AMQ are bloom filters. There are many other data structures that outperform bloom filters. For example, Morton Filters [5]. A bloom filter works by having a bit vector of size m and k hash functions, with $h_i(x) \in [0, m-1]$. When checking if an item exists or not, all k hash functions are evaluated, and we read the bits in the bit vector at $h_i(x)$. If one bit is zero, we can say that this element was not yet inserted. If all bits are 1 that element might have been inserted already, but it could also happen that the bits might have been set by other elements. To insert an element, just set the according k bits of the hash functions. As no bits get cleared, it is trivial why the false positive rate rises with an increasing number n of inserted clauses. We illustrated an example of a bloom filter in Figure 2.1.

2.4 Related Work

To our knowledge sharing clauses across different problems has not been done in distributed SAT solving.

The idea of clause strengthening, though, is in its core similar to ours. Clause strengthening has multiple names in academic literature, which include distillation [8], minimization [12] or vivification [14]. All of those follow the same principle. With clause strengthening, we try to improve the quality of clauses by checking if a subclause of a given clause may already be implied by the rest of the formula. If this is the case, we can swap the old clause with the new subclause and thus have shortened the length. Shorter clauses are most of

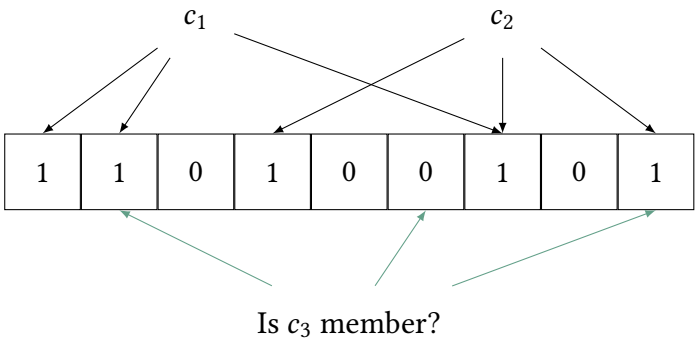


Figure 2.1: Example of a bloom filter with $m = 9, k = 3$. c_1 and c_2 were already inserted. c_3 will be correctly identified as not part of the set as bit 5 is not set.

the time preferred because in unit propagations they become unit more easily [12], which speeds up solving.

With our cross problem clause sharing approach, we also want to check if an externally shared clause is already implied by the rest of the local formula. If we can prove that an externally shared clause is redundant, we know that the clause is per definition valid in the given context and can be accepted into the local clauses. So there are genuine parallels between the two approaches.

Clause strengthening works the following way: Let $C = L_1 \vee L_2 \vee \cdots \vee L_l$ the strengthening candidate and $L_1 \dots L_l$ are ordered by some branching heuristic. A branching heuristic that was commonly used in the past was the MOMS heuristic [14]. But there are many others like the Jeroslow-Wang or (R)DLCS and (R)DLIS Heuristics [4]. The branching heuristic can have great impact on the success rate of the strengthening attempts. After the ordering, we perform the following checks and operations [14] (we provide a textual description in the paragraph below):

1. If for some $i \in 1 \dots l-1$ and $j > i$, $UP(F \setminus C \cup \neg C_i)$ deduces L_j,
   then $F \leftarrow (F \setminus C) \cup \{L_1 \vee L_2 \vee \cdots \vee L_i \vee L_j\}$

2. If for some $i \in 1 \dots l-1$ and $j > i$, $UP(F \setminus C \cup \neg C_i)$ deduces $\neg L_j$,
   then $F \leftarrow (F \setminus C) \cup \{L_1 \vee L_2 \vee \cdots \vee L_i \vee \cdots \vee L_{j-1} \vee L_{j+1} \vee \cdots \vee L_k\}$

3. If for some $i \in 1 \dots l-1$, $UP(F \setminus C \cup \neg C_i)$ deduces $\bot$ then extract a nogood C_l from the conflict using the First UIP scheme.
   If $C_l \subset C$ or $|C_l| < |C|$, then $F \leftarrow (F \setminus C) \cup C_l$
   otherwise $F \leftarrow (F \setminus C) \cup \{L_1 \vee L_2 \vee \cdots \vee L_i\}$

This in general leads to shorter clauses.

Now we want to give an intuition, what the steps do. We test, given the heuristic, all sub-clauses with only the first l literals. In check 1, we check if one of the literals not yet inserted in the sub-clause appears as a unit clause after unit propagation with the negated sub-clause. In this case, we know that this literal must also be true for the formula to be satisfiable. So we can learn the checked sub-clause and additionally add the literal.

If we would now redo the unit propagation with the (negated) clause and the (negated) additional literal, that literal would be assigned the opposite way and thus result in a conflict. This means that the new shorter clause is valid.

Check 2 is similar. Here, however, we find that one of the remaining literals is negated after the unit propagation. Therefore, we can conclude that this literal can be dropped in the shortened clause, as if we would add that literal, we would not get a new conflict.

In the last check, we find a conflict after the unit propagation. That means, every literal from the sub-clause is needed. We can still try to improve the clause with a conflict analysis, or accept the short clause, depending on what results in the shorter clause.

It is well acknowledged that the performances of solvers is usually greatly improved by preprocessing [14]. Usually, clause strengthening is also used as a preprocessor in SAT Solving and shows performance improvements. However, we have also seen an inprocessing approach [18] that is structurally very similar to the architecture of our External Clause Checker. This approach has a separate solving thread on the side and tries to strengthen learned clauses while solving. Wieringa and Heljanko [18] see speedup with their technique.

3 Our Approach

In this chapter, we present our contributions that enable sharing clauses across problems. In the first part, we present how we share data, like clauses, across problems. Section 3.2 explains how we validate clauses to avoid learning wrong information. At last, we talk about instances where we can skip this validation step.

3.1 Inter Job Communication

A subgoal in the course of Chapter 3 will be being able to exchange clauses between groups of similar jobs in the Mallob framework. In this section, we set the more general goal of sending arbitrary data between groups of related jobs. The data will be distributed in a ring structure. We opted for a ring structure as it was easy to implement, and we only expect small to moderate group sizes, so latency is not a big factor. We begin by explaining how the ring structure is build and afterward describe how data is being sent around the ring.

3.1.1 Ring Buildup

Only the root worker of a job, i.e. the job representative, will be part of a ring. We can make this simplification because the job representative will process every internally shared clause. For incoming messages, the job representative can quickly propagate the message to all children.

To signal to Mallob that jobs are part of the same group, the user can set a group ID for every job. Jobs with the same group ID will share their data. If the user did not specify a group ID, the software will work as without any cross-problem clause sharing.

To build a ring structure, all workers participate in a periodic all-reduction (once a second in our implementation). We chose this interval as it links to the default sharing interval of SAT jobs in Mallob. This recurring all-reduction makes it easy to handle jobs that will enter the system at a later point in time, while still having limited communication overhead. In this all reduction, every worker checks if it is the root worker of a job. If this is the case, it shares his group ID, his MPI rank, and if it is already part of a ring. If a worker is not a root worker of a job, he will participate in the reduction call but will not contribute anything. This information is aggregated in a tree-like fashion where always two datasets are merged, the resulting dataset is then merged with another merged dataset, and so on.

If there are multiple job representatives with the same group ID while aggregating data during the all-reduction, we distinguish different cases. If both job representatives are part of a ring or both job representatives are not part of a ring, just a random job

representative gets selected to be further passed on in the all-reduction. This helps with spreading the ring management load on multiple workers. If one job representative is part of the ring and the other is not, the ring job representative will always be further passed on in the all-reduction. Figure 3.1b illustrates this approach. The distinction between ring job representatives and non-ring job representatives is important, to not destroy any existing rings.

In the end, the aggregated data contains a group representative for every group ID. A job representative that wants to join a ring then can file a request to join the ring at the respective group representative.

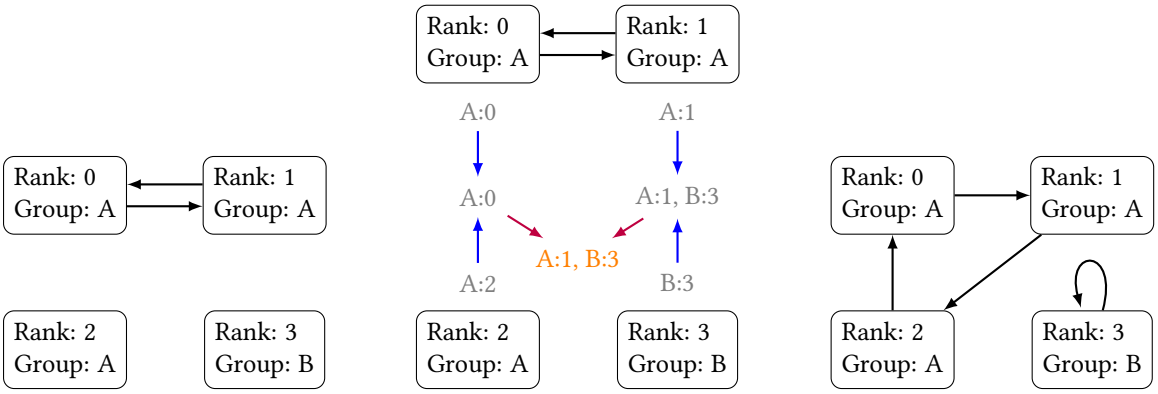
When every job representative has the list of all group representatives, group representatives, that are not part of a ring, just create a ring by themselves. Group representatives that are part of a ring do not have to do anything. Job representatives that are not part of a ring will submit a request to the representative for their group. The group representative sets its ring neighbor to the requestor and sends the requester their previous ring neighbor R. The requestor then sets R as his ring neighbor. Because it can happen that a group representative gets messages from other job representatives before he handled the data himself, we added a queue for such messages. That queue stores all incoming messages while a group representative is not ready. After the group representative processed the incoming data, all previous messages in the queue will get handled.

As an example for the ring creation algorithm, we will now elaborate Figure 3.1. Figure 3.1a shows the initial situation. The job representatives with ranks 0 and 1 are together in a ring. 2 and 3 have not yet participated in any all-reduction.

Figure 3.1b shows the process of sharing group representatives. The blue arrows describe the first aggregation step of the tree like all-reduction. On the left side, the datasets of 0 and 2 are aggregated. Both are in group A. 0 is part of a ring. Therefore, in this aggregation step, 0 is always passed on as the representative. In the aggregation step on the right, 2 and 3 are both in different groups. So both are passed on as representatives. The next aggregation step is shown in red. Here, 0 and 1 are both representatives of the same group and ring members. By random selection, 1 is passed on. This randomness can help to balance the load so that the same job is not always group representative. As 3 is the only representative for B, he also gets passed on. In the orange list are the group representatives. Each process now has access to this list.

Figure 3.1c shows the resulting ring structures. 2 was not part of a ring and makes a join request to 1. 1 sets his neighbor to 2 and sends 2 to his previous neighbor 0. 2 then puts his neighbor on 0. 3 sees that he is his own representative and therefore makes his own ring.

When a job is finished, the MPI process whose job finished remains a part of the ring. Note that in our current implementation, a ring will be broken once a new job with a different group ID is deployed on an MPI process which still relays ring messages from a former job. This is due to a one-to-one mapping in our code between an MPI process and its role in a ring. Since such a scenario is unlikely in the scope of our experiments, our solution is to rebuild all rings once every 20 seconds. For more general use cases of our approach, our implementation can be improved in the future to support an arbitrary number of message relays plus an active job on each MPI process.



(a) Initial situation before an all-reduction starts.

(b) The process of sharing ring representatives.

(c) The resulting ring structure.

Figure 3.1: Example of a ring-buildup.

3.1.2 Message Passing

Once job representatives are gathered into a ring, every job representative can send data into the ring at any time. In the case of clause sharing, a job representative would send clauses into the ring of similar jobs when internally shared clauses arrive at it. When the request comes to send data into the ring, additional metadata is created. The data and metadata are encapsulated in a *ring message*. The metadata contains the rank and group ID of the job that emitted the message. The rank is needed, so a ring message does not get passed around indefinitely but gets discarded after one round in the ring. The group ID is needed because the workers are not permanently bound to one specific job. So when a message is passed to a worker with rank i it could be that this worker is now a worker for a different job. So every time a ring message gets passed, it first needs to be verified with the group ID. If the verification fails, the message gets discarded.

If the verification passes, an arbitrary set action will get executed. We will call this action *Ring Action*. In the case of clause sharing, this action would be to check if the given clauses are applicable and if they are, include them in their own clauses. This action can be set at any time. It does not have to be set when a message arrives. If it is not set, then no action will be executed and the message will still be passed on. It makes sense, to set the action in the constructor of a Job. A quick overview of the message passing algorithm is laid out in Algorithm 2.

3.2 Clause Validation

When clauses are shared with a job representative, the ring action gets invoked. In the case of a SAT job, the Ring Action would pass the shared clauses to the clause checkers.

Within a SAT process, a SAT engine runs. In a SAT engine, there are multiple solver threads devoted to solving the local problem. Additionally, we added solver threads that are used only for the validation of external clauses. We will call these threads *External Clause Checkers* (ECC). The user can pass the parameter `numECCThreads` to determine how

Algorithm 2 Pass on ring message

```
Input: RingMessage r
  if r.group_id ≠ my_group_id then
      return
  end if
  if ringActionIsSet() then
      executeRingAction(r.data)
  end if
  if r.start ≠ my_neighbor then
      passOnRingMessage(my_neighbor, r)
  end if
```

many threads should be devoted to clause checking. The ECC threads only exist in the SAT engine of a job representative. By default, the ECC threads do not displace solver threads. But with the parameter `eccDisplaceSolver` the enable the displacement. If this option is active, and we start mallob with t threads per process and e ECC threads, the job representatives get $t - e$ solver threads and e ECC threads. All other processes get the full t solver threads. ECC threads take away resources from the solver threads and thus might also slow down the total solving time.

Clauses arrive at the SAT engine in the form of an integer vector. At first, they get disassembled. We hash every clause and depending on the hash of the clause it gets assigned to an ECC. This way we have a relatively even load balancing as we expect that over time all ECCs get the same amount of clauses with properties like LBD or length also averaging out. These clauses will get buffered in an internal buffer of every ECC to make the calls to an ECC non-blocking. We illustrated the integration of the ECC with the SAT engine in Figure 3.2.

Approximate Membership Query Filter Before we insert clauses into the local buffer of the ECC, we check with an Approximate Membership Query Filter if that clause was already shared. We do this because, for similar problems, we expect some clauses to be shared multiple times (from different jobs). We also check for every clause if all literals of the clause are valid variables of the local problem, otherwise the clause also gets discarded. Because at the creation of the External Clause Checker it is not known how many clauses will be shared, you have to use an AMQ that supports growing.

It is to note that because of the usage of AMQ, we might throw away some clauses that were not any duplicates. This, though, is a necessary memory-usefulness-tradeoff.

In our implementation we used Bloom filters which do not support growing by default. As a solution, we implemented a hierarchical AMQ that can also be applied to any other non-growing AMQ. We initialize the filter that is optimal for n members, and a maximum false positive rate of p (for one layer). After the insertion of n members, we allocate a new AMQ twice the size of the previous. New items will be inserted into this AMQ. When membership is queried, it returns true, if any of the AMQ likely contains the item. The doubling of the AMQ bit set results in a logarithmic amount of AMQ structures to be checked in a membership query.

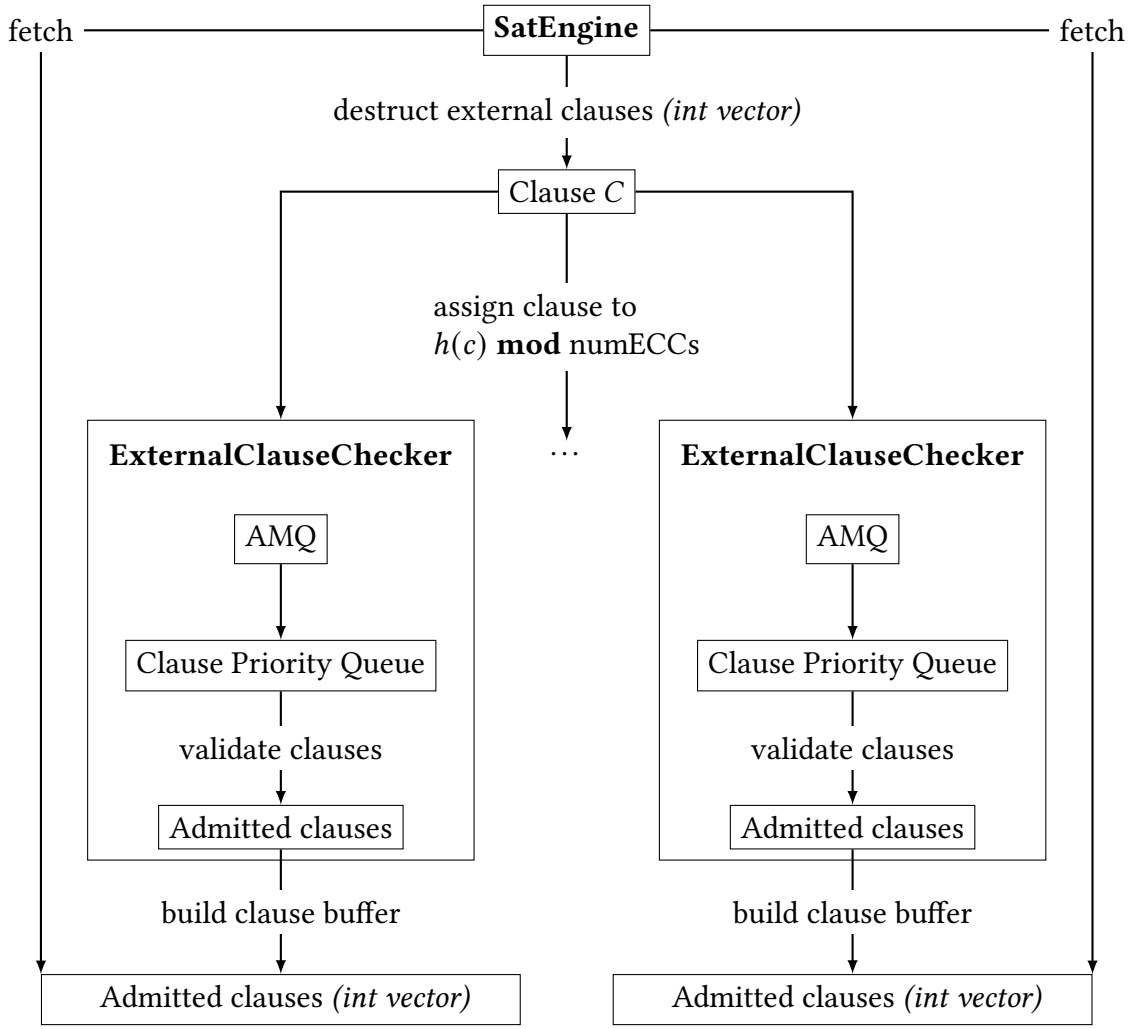


Figure 3.2: Integration of the External Clause Checker with the SAT engine.

We set the initial space n for members in our hierarchical AMQ to 2^{12}, as we did not have any example dataset where fewer clauses were shared. As for the maximum false positive rate, we chose $p = 0.01$. We did not encounter memory problems for this p, so there was no reason to check a lower value. Also, 1% of false positives is sufficient, as missing out from just a few shared clauses being discarded early before checking, does not harm the algorithm much.

It is to note that the more layers there are added to the AMQ, the worse the false positive rate gets. But even with 10 layers which can hold up to

$$\sum_{n=0}^{9} 2^{12+n} = 4\,190\,208$$

members, the expected false positive rate would still just be $1 - 0.99^{10} \approx 0.0956 < 10\%$.

We also implemented an approach with Morton filters, which should outperform classical Bloom filters [5]. But due to technical reasons (because we have to hash clauses and not integers) we had to fall back to use Bloom filters as a base AMQ for our hierarchical AMQ. Mallob already offered a Bloom filter structure that also supports clause hashing. In the case of Bloom filters and with $p = 0.01$ we need to allocate 9.6 bits per element to achieve our per-layer false positive rate [6].

Priority Queue / Fixed-size buffer It is expected to happen that the validation of clauses takes too long to check all external clauses that arrive at the ECC. This is especially true as the number of group members increases, since the results of one group member will be shared with every other group member. To limit the memory usage of the clause buffer, we implemented a fixed-size buffer with a maximum capacity of N clauses. For this buffer, we implemented the following item replacement strategies for full buffers:

- FIFO
- LIFO
- Prefer low LBD
- Prefer long clauses
- Prefer short clauses

All ECCs periodically check if there are new clauses in the buffer. Once there are clauses, one clause at a time gets checked. Here, the buffer acts as a priority queue. The most important clause of the currently used buffer, gets extracted. Depending on the buffer, there are different pros and cons. *FIFO* and *LIFO* don't require much overhead for inserting and extraction of clauses. *Prefer long clauses* has the advantage that long clauses tend to be verifiable more quickly but carry less useful information. *Prefer short clauses* is the opposite. Solving tends to take up more time, but short clauses tend to be more helpful in solving the local problem. The same is true for clauses with low LBD [18]. We will evaluate these buffers in Chapter 4.

Checking Clause Validity With the selection strategy of our chosen buffer, we now have a clause C to check. Given $C = L_1 \vee L_2 \vee \cdots \vee L_l$ for the external boolean formula F' we want to find out if C is also applicable for our local formula F . To do this, we can check if $F \wedge \neg C$ is unsatisfiable. We can check validity this way because if a clause C is recorded for F' we can interpret it semantically as: In order for F' to be satisfiable, at least one of the literals must be satisfied. When checking

$$F \wedge \neg C \quad = \quad F \wedge \neg(L_1 \vee L_2 \vee \cdots \vee L_l) \quad = \quad F \wedge \neg L_1 \wedge \neg L_2 \wedge \cdots \wedge \neg L_l$$

we have the following thought process: If at least one of the literals L_i has to be `true` for F' to be satisfiable, then if that clause is applicable for F, also at least one of the literals L_i has to be `true` in F. So we test what would happen if all literals were `false`. For similar problems, we expect that then the formula is no longer satisfiable. If this is the case, we validated that C is applicable for F. However, if the formula is satisfiable, we know that this clause is not applicable to our problem. We have listed a overview of the algorithm in Algorithm 3.

The usage of incremental SAT solvers, can be very useful in this context here. For every negated literal of the negated clause we can set an assumption in the solver and thus can reuse the same solver for all clauses and do not have to read in the formula multiple times.

Now we can also see why the validation of long clauses generally tends to be faster. The longer the clause C , the more assumptions we can extract from $\neg C$, and the easier the problem is because $|C|$ truth variables are already fixed from the beginning of the validation process. However, this does not mean that this always applies. Some short clauses may be validated quicker by having well-set literals.

Because completely checking if the given clause is applicable takes too long, as this sub-problem is also NP-complete, we only check with unit propagation if we can find a contradiction. If we find one, we can set the clause status as applicable, as we have seen before. If we find none, that could be either due to that the clause is not applicable to the problem because the formula is satisfiable or because we did not find the contradiction with the unit propagation. This may lead to discarding some applicable clauses but in general, saves time.

Timeout The validation of any clause should not take too long, to not clog up the system. With an unfavorably chosen clause, the solver might get stuck for several seconds and many (maybe more promising clauses) get discarded as a result. This is why we implemented a timeout for the ECC solvers. We will discuss sensible values for the timeout in Chapter 4.

Another interesting approach would be to determine how many clauses arrive in one second and set a minimum percentage of the clauses we want to validate. Depending on that, we can set the timeout dynamically. Example: 500 clauses get shared every second with a solver. The ECC wants to check at least 10% of clauses. Then the timeout should be set to $1000\text{ms}/(500 * 0.1) = 20\text{ms}$.

Clause strengthening Furthermore, we integrated clause strengthening, into our work. We have explained the principle behind clause strengthening in Section 2.4. As we have already seen in Paragraph 2.1, some good redundant clauses, i.e. clauses that are implied by other clauses, can greatly speed up problem-solving. But we have also seen that too

Algorithm 3 Validate clauses

```
Input: Formula F , Buffer B of clauses to check
  while B not empty do
      C ← B.extract()
      assumptions ← ∅
      for Literal L in C do
          assumptions.add(¬L)
      end for
      result ← check(F, assumptions)          ▷ Timeout if validation takes too long
      if result == UNSAT then
          validated_clauses.add(C )
      end if
  end while
```

many of these redundant clauses can slow down solving attempts [14]. This could also be a problem with our approach, since we have the internally shared clauses on the one hand, and *additionally* the externally shared clauses on the other hand. Both sources of clauses combined may be harmful to the performance. We tried to use clause strengthening as a solution. Here we spend more time on a single clause and try to improve it. Because of the clause strengthening attempts, fewer external clauses will overall be accepted, and those that are accepted are, on average, of higher quality.

Due to technical limitations, we could not implement the strengthening algorithm from Wieringa et al. [18] one to one, but had to strip down the algorithm to fit our needs. We will only look at the third case of the algorithm, where a sub-clause $C' = L_1 \vee L_2 \vee \cdots \vee L_i$ leads to the unsatisfiability of the problem. If this is the case for an $i < l$ we successfully have shortened - or strengthened - the clause. As already stated in Section 2.4 the order of the literals can have a great impact on the success rate of the clause strengthening. We implemented the Jeroslow-Wang Heuristic [10] to sort the literals of C before performing the solving tries. We illustrated this algorithm in Algorithm 4. The areas that differ from the default clause validation are highlighted.

It would also be possible to use the ECC just as a clause strengthening thread, with no cross problem clause sharing at all. This was a functionality mallob did not have before. We did not further experiment with this.

3.3 Identical Problems with different Assumptions

For some solving techniques, we have to solve multiple identical instances where only the assumptions differ between these instances. Because all clauses stem from the same instance, we can guarantee that clauses are applicable between instances and we can disable the verification step. This way, we can use the full processing power for the solving threads, while also benefiting from cross-problem clause sharing.

One such technique is the Cube and Conquer approach. Cube and Conquer consist out of two phases. At first, we partition the problem into many sub problems, so-called *cubes*.

Algorithm 4 Validate clauses with clause strengthening

```
Input: Formula F , Buffer B of clauses to check
  while B not empty do
    C ← B.extract()
    sortLiterals(C)                          ▷ Different literal sorting strategies possible
    assumptions ← ∅
    for Literal L in C do
      assumptions.add(¬L)
      result ← check(F, assumptions)▷ Check inside add-literal-loop, instead of after
      if result == UNSAT then
        validated_clauses.add(C )
        break
      end if
    end for
  end while
```

Afterward, a conflict-driven solver tackles these sub problems [9]. We can realize the partitioning by setting different assumptions for the same instance. For example, to partition a problem into 4 cubes, you could add the assumption $[1, 2]$, $[1, -2]$, $[-1, 2]$, $[-1, -2]$ to 4 identical instances of the problem. These 4 instances could just be placed in group and be solved together with cross problem clause sharing. Normally, a heuristic is used to decide on which literals to "divide" the problem.

Another application where several problems with the same clauses but with different assumptions are solved is the improvement of found plans in automated planning, e.g., in hierarchical planning [16].

4 Evaluation

In this chapter, we will evaluate our cross problem clause sharing approach. We start off by giving some notes and details on our implementation in Section 4.1 and presenting our evaluation setup in Section 4.2. We will then optimize the parameters of our approach. And lastly, we will continue with an in depth comparison of our approach to the baseline without cross problem clause sharing on multiple datasets.

4.1 Implementation

We will now elaborate on some implementation details. For a closer look, our source code is available at `https://github.com/SamuelBorn/mallob`. As a SAT solver for the clause validation, we used CaDiCal [2].

Path of Externally Shared Clauses To illustrate the path of externally shared clauses through Mallob, we made Figure 4.1. ForkedSatJob 0 shares clauses. ForkedSatJob 1 is the next ring member after 0. Normally, 1 would also send the clauses to its next ring member, and so on. For readability purposes, we abstracted this away in this illustration. Furthermore, we need to pass clauses through an Inter-Process Communication interface when passing messages from an MPI process to the SAT solvers, as SAT solving is performed in a separate sub-process.

It is to note that the clauses have to take the detour over the worker. This was a technical limitation by mallob. Mallob did not allow for multiple callbacks for one process, but one worker could manage multiple jobs. In future versions of mallob this limitation will be removed. This was just not yet reflected in our codebase.

Prevent Polling To prevent constant polling, if there are new clauses to check, the External Clause Checker has a conditional variable that puts it to sleep when there are no clauses to check in the buffer. As soon as the engine submits new clauses, it gets wakened up again. This ensures that no CPU resources are wasted on polling.

Enabling Clause Checking in Mallob To specify the group of a job, add the field “group_id” to the job description. For ease of use, string IDs will just be mapped to hidden integers IDs, and we store a map to translate between string and int group_ids. If no group_id is passed, there will be no cross problem clause sharing. We also added an easy switch to disable externally shared clauses getting checked for validity (useful for identical problems with different assumptions). This can be set with the field “check_external”. If this field is not set, the default behavior is to check external clauses. You can see an example job description in Figure 4.2.

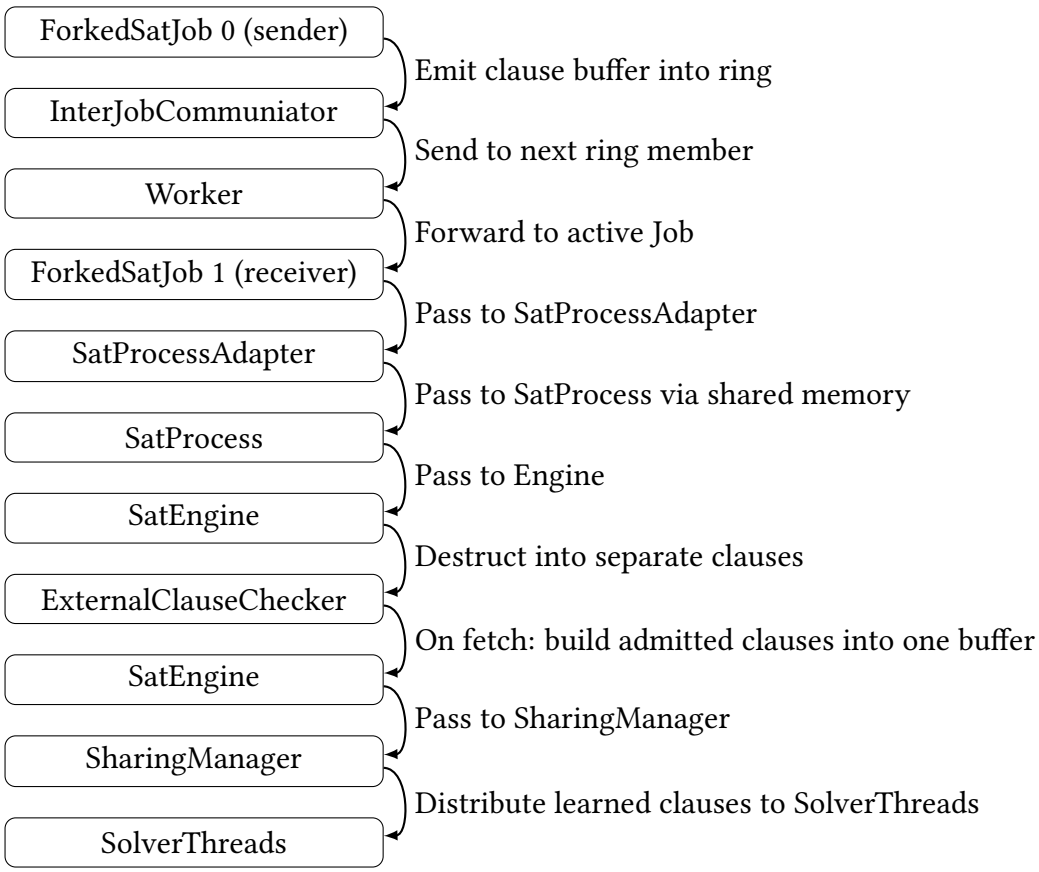


Figure 4.1: The path of externally shared clauses through Mallob.

```
{
    "application": "SAT",
    "user": "admin",
    "name": "test-job-1",
    "group_id": "group-with-a-name",
    "check_external": true
}
```

Figure 4.2: Example Mallob job description to enable cross-problem clause sharing for the given job.

ID	OS	CPU	RAM	Caches (per Socket)
pc132	Ubuntu 20.04	2 x Intel Xeon E5-2683 v4 2.1 GHz 2x16-core (+ 2x16-HTcore)	512 GiB	40 MB Cache
pc133	Ubuntu 20.04	AMD Ryzen 3950X 4.7 GHz 1x16-core (+ 1x16-HTcore)	64 GiB	L1: 1024KB L2: 8MB L3: 64MB
pc136	Ubuntu 20.04	AMD EPYC Rome 7702P 2.0-3.35GHz 1x64-core (+ 1x64-HTcore)	1 TiB	256 MB L3 Cache
pc137	Ubuntu 20.04	AMD EPYC Rome 7702P 2.0-3.35GHz 1x64-core (+ 1x64-HTcore)	1 TiB	256 MB L3 Cache
pc139	Ubuntu 20.04	Intel Xeon Gold 6314U 2.3GHz 2x32-core (+ 2x32-HTcore)	512 GiB	512 MiB

Table 4.1: Hardware specifications of our machines.

4.2 Experimental Setup

In this section, we will describe our experimental setup. In particular, we will describe in which environments and with which test data we performed our tests. We will also explain which parameters of our approach we have tuned.

4.2.1 Environment

We ran tests on different hardware. For every result, we will mention on which computer the tests ran. In Table 4.1 we listed the specifications of the computers.

4.2.2 Tuning Parameters

We will tune our parameters in a waterfall like fashion. Starting with a basic setup of parameters, we will optimize for the first parameter. Thereafter, that parameter is fixed, and we will proceed with the next parameter and so on. We will optimize the following parameters in the given order:

1. *Group size*: How large should groups be? More group members lead to more clauses being shared with a Job. But because not all clauses can be checked, some important clauses may be tossed.

2. *ECC Timeout*: What is an ideal timeout for an ECC verification attempt? Long timeouts may verify important clauses, but deprive other clauses of solving time. We will test timeouts between 10ms and 100ms.

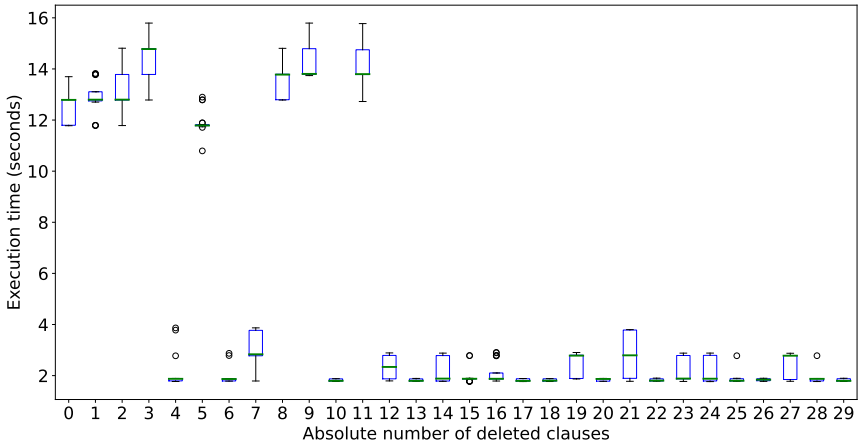


Figure 4.3: Result of deleting N random clauses. The solving time fluctuates, and it quickly gets trivial to solve instances. *(pc132)*

3. *Clause Priority Queue*: What is an ideal ordering strategy for the clause buffer. This determines which clauses get checked. We will evaluate every strategy mentioned in Paragraph 3.2.

4. *Clause Strengthening*: We will determine if it is worth to enable our simple clause strengthening.

5. *Number of ECC threads*: At last, we will test multiple configurations for the number of solver threads and ECC threads.

4.2.3 Instances

As sharing only makes sense for similar problems, we needed similar problems to check the validity of cross problem clause sharing. We looked at multiple ways to create such problems.

Delete N random clauses from an existing formula This approach did not work reliably. Removing just one key clause can make a problem trivial to solve. For our testing purposes, we wanted the solving attempts to always take a similar amount of time. This especially effects satisfiable instances. We visualized this behavior in Figure 4.3. We tested the execution time of a randomly generated 3SAT instance with 300 variables and 1280 clauses when deleting 0–29 clauses (of the same 3SAT instance).

Add N random clauses to an existing formula This approach suffers from the same shortcomings as the last one. Solving might get trivial, if an unfavorable clause gets added. This especially effects unsatisfiable instances.

Generate random instances, with an overlapping core Another approach was to create random instances, that share a common set of clauses. We created the instances, using a script that already existed in the Mallob source code files, which provides the algorithm generate_random_3SAT_clauses(nums_vars, num_clauses). In Algorithm 5 we illustrated how we generated those instances. The magic number 4.17 is a result from [11], which states that a random 3SAT formula is on average maximally difficult when there are 4.17 times more clauses than there are variables. With this approach, we also have the same shortcomings as in the previous approaches.

Algorithm 5 Generate random similar 3SAT instances

Input: Number of Variables V, Common core percentage p, Formulas to create N
```
common_num_clauses ← [4.17 · V · p]
varied_num_clauses ← [4.17 · V · (1 − p)]
common_clauses ← generate_random_3SAT_clauses(V, common_num_clauses)
formulas ← {}
for i in [0..(N − 1)] do
    varied_clauses ← create_random_3SAT_clauses(V, varied_num_clauses)
    formulas.add(common_clauses ∪ varied_clauses)
end for
```

Planning instances We finally decided to use planning problems. For a given planning problem, we encode that problem, where a maximum of k actions are allowed, as SAT instances. That leads to a sequence of increasingly difficult unsatisfiable instances until a first instance with x actions is satisfiable. We were able to create such instances using Madagascar [15]. Madagascar is an implementation of the SAT based techniques for planning. Besides solving planning problems, Madagascar can also output the respective CNF for each planning step. The CNF for k actions is a prefix of the CNF for $k+1$ actions, and identical variables also share identical meanings across these instances. This makes those instances ideal for us. When it is not import to find the shortest solution, but it's just a matter of knowing if a reasonable short plan exists, we can run Mallob with k jobs $X,\ X+1, \ldots, X+k$ and cross problem clauses sharing enabled. They can then share clauses with each other and benefit from each other's partial results. For planning instances, we sourced the classical tracks from the International Planning Competition 2018.

We filtered out instances, that took less than 20 minutes to solve. On the other hand, we also did not look at instances where Madagascar was unable to convert the planning instances into SAT instances within five minutes. For the remaining instances, we sampled random instances with varying difficulties.

With this, we have "killed two birds with one stone". On the one hand, we have numerous instances with reliable execution times. On the other hand, we have found a potential use for cross-problem clause-sharing in the real world, so we don't have to work only on synthetically created instances.

We will test planning instances mainly with a sliding window of similar instances. There will always be j Jobs working on the instances $I_i, I_{i+1}, \ldots, I_{i+j-1}$ in parallel. As soon as the job with the instance I_i finishes, it gets replaced by another job for the instance I_{i+j}. For

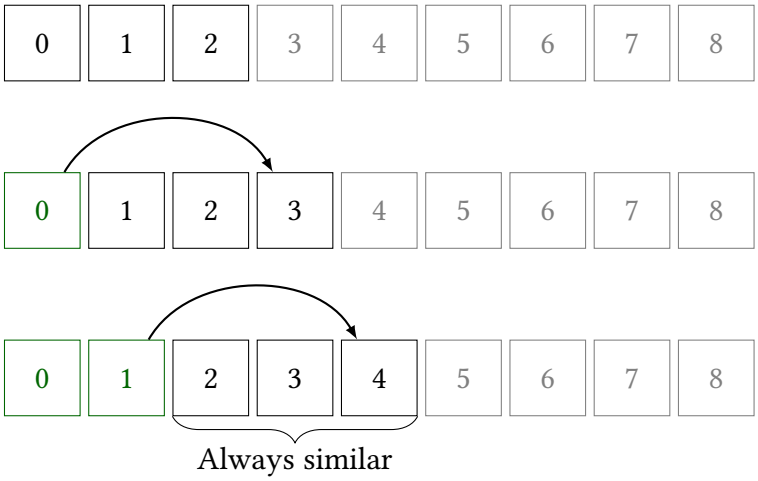


Figure 4.4: Sliding window test instances. Sliding window size = 3. Black nodes are instances currently being worked on. Green nodes are completed instances. When an instance is solved, the next not yet solved or solving instance gets solved.

$x < y$ we expect the job with instance I_x to finish first. This is because it started before y and because we chose instances, that are just getting harder until a break point. But we will not reach this breakpoint in our selected instances. So when x mostly finishes before y this ensures that the group will always stay similar. But even a little bit of scattering of the jobs is not bad and will just lead to more clauses being rejected. We illustrated the sliding window in Figure 4.4. For the instances we looked at, approximately a constant 200,000 new clauses got added for each level. So the higher levels should be more similar, because they share a larger core of clauses.

4.3 Evaluation of Clause Sharing

At first, we will determine the optimal parameters as discussed in Subsection 4.2.2. After that, we will compare the performance for some instances.

We will use "PxT" as a shortcut for a system configuration with P processes and T threads per process.

4.3.1 Parameter tuning

Our base parameters will be 64x1 configuration, a timeout of 50ms, preferring low LBD clauses, clause strengthening being disabled, and one additional ECC thread for the job representatives.

Optimal group size The test results for determining an optimal group size are illustrated in Figure 4.6a. The legend of the figure can be interpreted the following way:

- *sharing x*: Both local and external clause sharing is enabled. There are always x active jobs of the same (and only) group.

- *no sharing x*: Only local sharing within a job is enabled. There is no inter job communication. There are always x active jobs.

For instances with sharing disabled, we just set the number of active jobs to the group size. We tested up to 5 group members with both sharing enabled and disabled. The largest discrepancy between instances with sharing enabled/disabled was for a group size of two. Also, with a group size of two, the execution time was the lowest while reaching the most number of instances solved. This result is not transferable to other contexts, though. Our experimental setup encourages small groups, because consecutive instances are more similar. That means, in our test setup, larger groups results in less similar group members. This leads to an overall worse performance.

Optimal ECC Timeout The test results for an optimal ECC solver timeout are illustrated in Figure 4.6b. Although the differences are small, a timeout of 20ms consistently performed the best. The timeout did not have a large impact because the many clauses get validated in less than even 10ms, which was our lowest timeout.

It is also interesting to see the distribution in which processing step clauses were processed. We have measured this, for the instances of `snake:p01` with horizons 35, 36 and 37, a group size of 3, the Min LBD buffer, and stopped as soon as one job finished. The first job finished, depending on the timeout and test run, between 420sec and 480sec. We ran the tests on 16x1 configuration, and one additional ECC thread for the job representatives. Sample results are visualized in Table 4.2. The exact numbers have varied depending on the test run.

As expected, we can observe that the amount of clauses that get rejected because of a validation timeout increases with a reduction of the timeout. It is also interesting to note that in most tests, ~60% of all clauses were caught by our hierarchical AMQ. Furthermore, we can observe that the highest rate of verified clauses is with a timeout of 20ms (green cell). This also supports our findings from Figure 4.6b, that suggests a timeout of 20ms performs the best.

We observed that with no timeout given, the average time a clause takes to be verified is about 15ms. This also explains why 20ms is a good timeout: we get to check most clauses while not having to waste too much time if a clause cannot be validated in a reasonable amount of time.

It is interesting to notice that there were no clauses in our test results. This is because the test was performed on an UNSAT instances, and we cannot disprove the validity of a clause in UNSAT instances. This is correct, as additional clauses only make the problem "harder" to satisfy, which only supports the unsatisfiability of the given problem.

On the other hand, proving that a clause is invalid can mostly not be done in the given timeframe, and clauses that might be rejected will timeout first, before being flagged as invalid.

Optimal Clause Priority Queue The test results for the optimal Clause Priority Queue are illustrated in Figure 4.6c. Again we find very close test results. Generally we can observe though that the minimum LBD buffer performs the best. This is also consistent with the results of the related work, which has shown that learned clauses with high LBD values are not very useful for solving practical SAT instances [12]. We can also observe that the

	Full buffer	AMQ	Verified	Rejected	Timeout
10ms	9.69%	59.91%	24.52%	0%	5.8%
20ms	10.23%	59.81%	28.12%	0%	1.83%
50ms	11.14%	62.1%	25.9%	0%	0.84%
100ms	14.6%	62.92%	21.82%	0%	0.67%

Table 4.2: How clauses get processed for ECC timeouts of 10ms, 20ms, 50ms and 100ms. *(pc139, 16 cores)*

maximum clause length buffer typically performed the worst. So the added bonus from being able to validate more clauses did not compensate for a worse average quality of the clauses.

Enabling Strengthening We can see in Figure 4.6d that our clause strengthening approach clearly did perform worse than the default with no strengthening. However, this might also be due in large part to our only "half" implementation of the clause strengthener, which does not take advantage of all the strengthening possibilities. We have observed that only $< 1\%$ of the clauses will be reduced in size. So for the remaining clauses we just waste computing resources, until we accept the full sized clause.

Multiple ECC threads We tested a varying number of ECC threads with the `nurikabe:p08` instances on both 16 processes with 4 threads per process and 8 process with 8 threads per process. Figure 4.6e compares the solver-displacing and the "additional" strategy for the ECC threads. The differences are marginal. We zoomed in to better see the results of the plot. The not visible part of the plot is very similar and not worth comparing the large dataset. The results become a bit more clear if we look at the solving times of a single instance. We can see such results in Figure 4.5.

Generally, we can observe that adding too many ECC solver threads harms the performance. We also observed that it is better to have additional threads for the ECC and not displace any solver threads. We also did the test for a configuration of 8x8. The results are illustrated in Figure 4.6f. Here too, the differences are marginal. One to two ECC threads seemed to perform the best.

It is also interesting to note how the number of clauses that get accepted depending on the number of ECC threads. We tested this with the `snake:p02` instances and observed the program after 100 seconds with 1-5 additional ECC threads with a 8x8 configuration. The number of solver threads were not changed. Sample results can be found in Table 4.3. The exact numbers have varied depending on the test run. We observed an increase in the number of accepted clauses when increasing the number of ECC threads. This increase is, as expected, not linear, as the number of processing elements devoted to a process stayed the same throughout the tests and the ECC threads just gained, proportionally to the solver threads, more time.

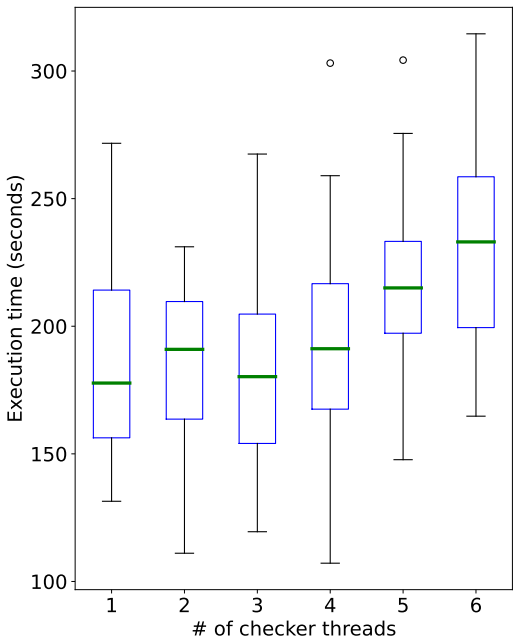
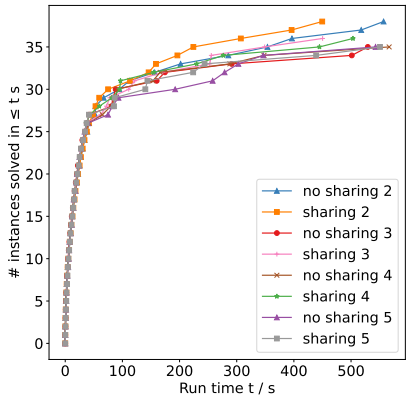


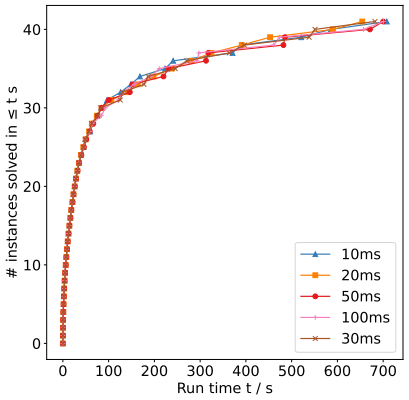
Figure 4.5: Experimental results for multiple non-displacing ECC threads.

# of ECC threads	# of accepted clauses
1	16451
2	20112
3	28474
4	29422
5	38404

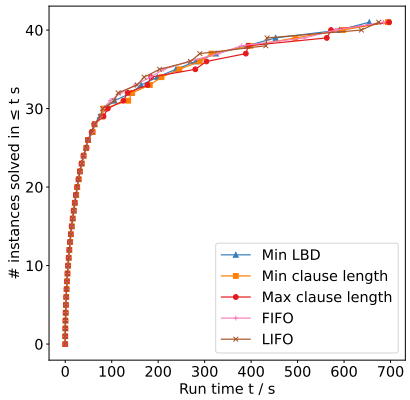
Table 4.3: Number of accepted clauses, depending on the number of ECC threads.



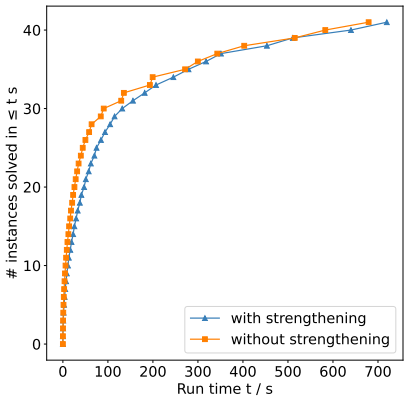
(a) Performance for different group sizes *(pc137, 32 cores)*



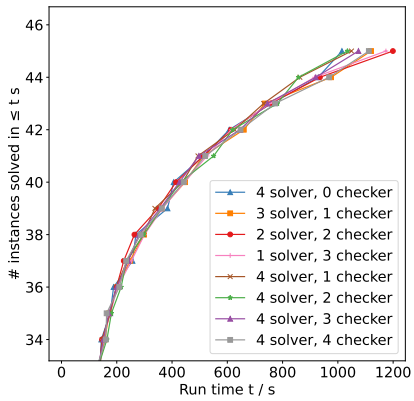
(b) Performance for different timeouts of the ECC. *(pc137)*



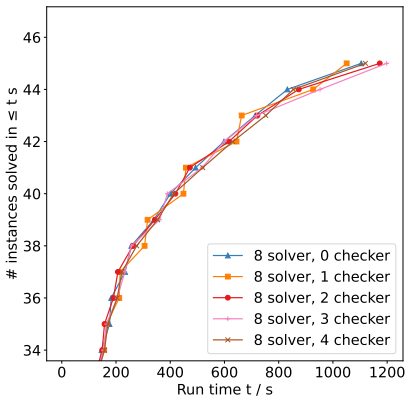
(c) Performance for different clause priority queues. *(pc137)*



(d) Performance with and without clause strengthening. *(pc137)*

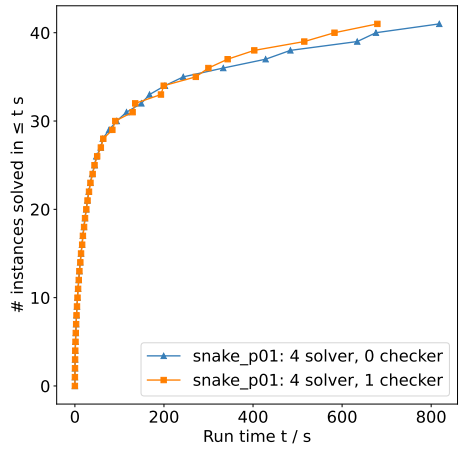


(e) Performance for 16x4, with 0-4 additional and displacing ECCs. *(pc137)*

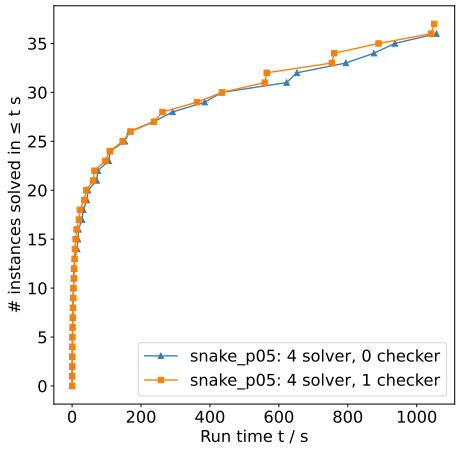


(f) Performance for 8x8, with 0-4 additional ECCs. *(pc137)*

Figure 4.6: Performance charts for parameter tuning.



(a) Example plot, where our approach slightly outperformed the baseline.



(b) Example plot, where our approach slightly outperformed the baseline.

Figure 4.7: Visualization of some test results.

4.3.2 Comparing results

Depending on the instance, when comparing our clause sharing approach to runs with no cross problem clause sharing, we either see a small speedup or no change in performance.

We have written down all our test results in Table 4.4. "CPCS" stands for cross-problem clause sharing. T_n describes the time when the last solution finished in seconds. T_{n-1} the penultimate finish time and so on. The solution times of the first instances are usually similar enough and propagate speedups to the last instances, so we don't show them here. We used a 8x4 config, with an ECC timeout of 20ms, the minimum LBD buffer, and one additional ECC thread for the solvers. Figure 4.7a and Figure 4.7b show examples where our approach slightly outperformed the baseline.

4.4 Evaluation for Identical Problems with different Assumptions

At last, we want to take a look at identical problems with different assumptions. Specifically, we looked at the cube and conquer approach. We tested our approach on the instances, that can be found in the Cube and Conquer GitHub Repository (`https://github.com/marijnheule/CnC`) from Heule et al. [9]. We created "good" cubes with Heules `march_cu` solver. The results can be seen in Figure 4.8. With most instances, we can observe a statistically significant speedup of ~10% when we disable clause validation.

Furthermore, for the given instances, both cube solvers (with and without cross-problem clause sharing) cleary outperformed the default "mono" way of solving the instances.

Problem	CPCS enabled	# solved in $t < 1200s$	T_{n-3}	T_{n-2}	T_{n-1}	T_n
data-network: p01	No	17	45.36	106.83	273.81	733.14
data-network: p01	Yes	17	44.52	117.99	257.57	742.98
data-network: p02	No	46	1083.13	1199.77	-	-
data-network: p02	Yes	48	1017.33	1100.82	1123.38	1189.64
data-network: p03	No	14	15.64	43.59	149.78	503.38
data-network: p03	Yes	14	15.47	44.39	149.79	521.13
data-network: p05	No	17	72.59	169.34	361.72	751.35
data-network: p05	Yes	17	73.71	166.72	364.11	761.17
data-network: p07	No	17	17.50	36.36	68.39	329.25
data-network: p07	Yes	17	18.88	36.62	59.29	332.97
nurikabe: p05	No	46	537.04	691.66	814.36	1046.26
nurikabe: p05	Yes	46	560.20	720.35	869.80	1070.43
nurikabe: p07	No	48	590.00	760.71	857.35	1163.15
nurikabe: p07	Yes	48	566.83	710.22	854.36	1141.76
snake: p01	No	37	374.36	498.23	765.26	1048.36
snake: p01	Yes	37	403.76	488.24	670.63	930.40
snake: p02	No	41	506.16	601.32	743.90	1104.74
snake: p02	Yes	41	541.94	575.66	860.70	941.45
snake: p03	No	42	460.62	572.45	748.70	-
snake: p03	Yes	43	516.98	685.02	867.20	1154.26
snake: p05	No	36	875.76	936.53	1057.30	-
snake: p05	Yes	37	760.43	889.13	1042.17	1050.48
snake: p07	No	39	563.97	745.21	924.72	-
snake: p07	Yes	40	637.25	639.39	950.49	1001.69
settlers: p01	No	46	878.68	956.73	1006.21	1090.71
settlers: p01	Yes	45	963.41	1042.27	1120.22	-
settlers: p02	No	17	12.94	34.04	84.43	686.87
settlers: p02	Yes	17	12.18	34.39	92.93	783.32
settlers: p03	No	16	22.22	52.30	164.36	-
settlers: p03	Yes	17	24.01	50.12	162.06	1121.35
settlers: p05	No	17	12.84	25.43	58.45	208.32
settlers: p05	Yes	17	11.95	26.10	58.05	234.35
settlers: p07	No	18	16.19	42.97	91.52	564.73
settlers: p07	Yes	18	15.23	43.99	91.43	581.55

Table 4.4: Comparison of our approach for different problems. *(pc137, 8x4+1ECC)*

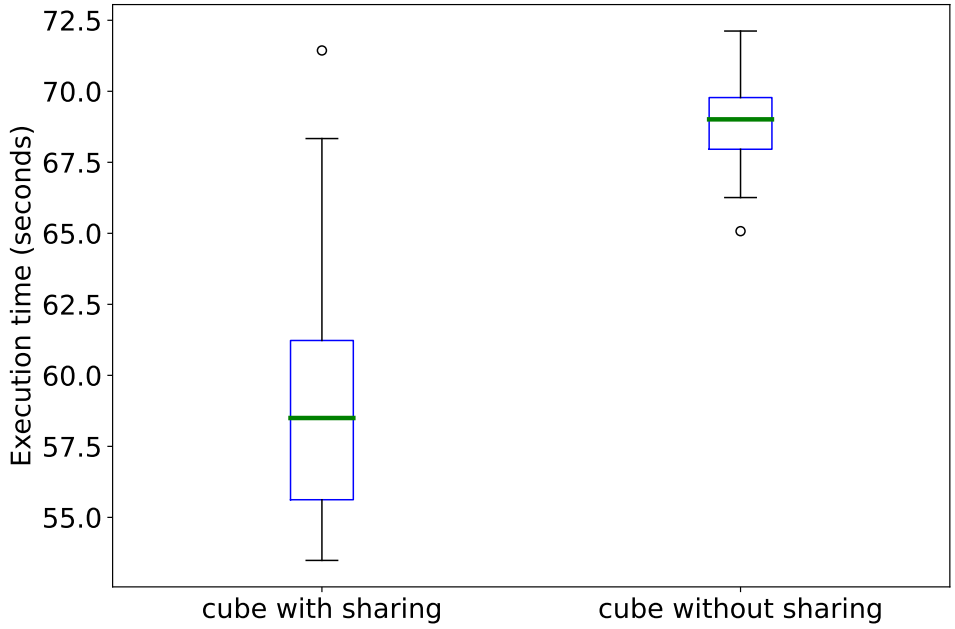


Figure 4.8: Results for cube and conquer solving. *(i10pc137)*

5 Conclusion

In this thesis, we have extended the traditional clause sharing of parallel SAT solvers by also sharing clauses between different problem instances.

We created a general communication framework on top of Mallob that enables the user to create groups of jobs, which then can share information which each other over a ring structure for every group. In the case of SAT solving, these jobs share internally shared clauses with the job representative of problems.

For problems that are not identical, we implemented a clause validation approach that is similar to the incomplete redundancy check of clause strengthening. With some instances, we could see a speed-up. Other instances though were not affected in the execution time. Our approach performed better on identical instances, where the validation step can be skipped. Here we observed a statistically significant speedup.

However, there are limiting factors that question the practicability of our approach. Firstly, the achieved speed-ups *with clause validation* were on average not very significant. Secondly, the number of instances in which our approach provides an advantage is limited, as we require very similar instances for our approach to work effectively. The similarity of the problems is proportional to the performance.

5.1 Future Work

Last but not least, we want to talk some possible extensions and improvements of our work.

Additional applications We primarily performed our tests on planning instances. It would be interesting to also consider additional applications. For example, similar instances can also occur in the context of software verification. Here different effects might be observable, and our approach might perform better.

Detect Similar Instances At the moment, the user has to manually specify which instances are similar. It could be possible to add a system, that automatically detects similar problems. One simple way to achieve this would be to simply take the "diff" between instance files and consider them equal if their diff is below a certain threshold.

Another approach would be to check for approximate graph isomorphism over the implication graph of the instances. This approach would remove the need for variables to be named the same. After an isomorphism is confirmed, we need to transform the instances into a common format, so that similar variables are called the same.

An extreme case where approximate graph isomorphism would outperform this would be two identical instances, where every $C = L_1 \vee L_2 \vee \cdots \vee L_l$ of one problem would be mapped to $C' = (L_1 + 1) \vee (L_2 + 1) \vee \cdots \vee (L_l + 1)$. Here, a diff based approach would

return a similarity of 0% while a graph isomorphism approach would return 100%. In such cases, we would have to build an additional translation layer so variables that are called the same also have the same meaning. This translation layer could simply be another step in the ring action.

However, approximate graph isomorphism is very expensive to compute, and itself NP-complete. It would have to be tested whether such a calculation effort would be worthwhile.

Use Internally Shared Clauses In the current implementation, internally shared clauses do not get sent to the ECC. It might be a good idea to also send them to the ECC and accept them without checking. The internally shared clauses could speed up future validation attempts. But as with every kind of clause sharing, too many shared clauses might also clog up the system.

Use Unknown ECC results When checking a clause for validity, we just use unit propagation because instead of a full solving attempt to save time. This leads to some validation attempts returning *unknown* because within the unit propagation the solver could neither verify satisfiability nor unsatisfiability. Here it would be interesting to further explore the clause. One could say that if the clause LBD reaches a certain threshold, then extra time will be spent on that clause. Again, it would have to be tested if this would be worth it.

Improve Hierarchical AMQ At the moment, we use bloom filters as the base AMQ for our hierarchical AMQ. There are alternatives though that outperform bloom filters on false positive rate / bits per element [6]. To reduce the memory consumption of mallob one idea would be to change out the base filters of our hierarchical structure. As previously noted, we took a look at Morton filters [5].

Improve Clause strengthening As we have seen in Subsection 4.3.1, our clause strengthening did perform poorly. It would be interesting to try an improved strengthening algorithm.

Bibliography

[1] Tomáš Balyo, Peter Sanders, and Carsten Sinz. "HordeSat: A Massively Parallel Portfolio SAT Solver". In: *Theory and Applications of Satisfiability Testing – SAT 2015*. Ed. by Marijn Heule and Sean Weaver. Cham: Springer International Publishing, 2015, pp. 156–172. ISBN: 978-3-319-24318-4.

[2] Armin Biere et al. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo et al. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[3] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[4] Tomáš Balyo Carsten Sinz. *Practical SAT Solving*. https://baldur.iti.kit.edu/sat/files/2018/l05.pdf. [accessed 12.3.2023]. 2018.

[5] Peter C Dillinger et al. "Fast succinct retrieval and approximate membership using ribbon". In: *arXiv preprint arXiv:2109.01892* (2021).

[6] Bin Fan, David G Andersen, and Michael Kaminsky. "Cuckoo filter: Better than bloom". In: *USENIX Programming* 38.4 (2013), pp. 36–40.

[7] 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.

[8] Hyojung Han and Fabio Somenzi. "Alembic: An efficient algorithm for CNF preprocessing". In: *Proceedings of the 44th annual Design Automation Conference*. 2007, pp. 582–587.

[9] Marijn JH Heule et al. "Cube and conquer: Guiding CDCL SAT solvers by lookaheads". In: *Hardware and Software: Verification and Testing: 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers 7*. Springer. 2012, pp. 50–65.

[10] Robert G Jeroslow and Jinchang Wang. "Solving propositional satisfiability problems". In: *Annals of mathematics and Artificial Intelligence* 1.1-4 (1990), pp. 167–187.

[11] Scott Kirkpatrick and Bart Selman. "Critical behavior in the satisfiability of random boolean expressions". In: *Science* 264.5163 (1994), pp. 1297–1301.

[12] Mao Luo et al. "An effective learnt clause minimization approach for CDCL SAT solvers". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 2017, pp. 703–711.

[13] Joao P Marques-Silva and Karem A Sakallah. "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521.

[14] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. "Vivifying propositional clausal formulae". In: *ECAI 2008*. IOS Press, 2008, pp. 525–529.

[15] Jussi Rintanen. "Madagascar: Scalable planning with SAT". In: *Proceedings of the 8th International Planning Competition (IPC-2014)* 21 (2014), pp. 1–5.

[16] Dominik Schreiber, Damien Pellier, Humbert Fiorino, et al. "Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 29. 2019, pp. 382–390.

[17] Dominik Schreiber and Peter Sanders. "Scalable SAT Solving in the Cloud". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2021, pp. 518–534. DOI: 10.1007/978-3-030-80223-3_35.

[18] Siert Wieringa and Keijo Heljanko. "Concurrent clause strengthening". In: *Theory and Applications of Satisfiability Testing–SAT 2013: 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings 16*. Springer. 2013, pp. 116–132.

[19] Lintao Zhang et al. "Efficient conflict driven learning in a boolean satisfiability solver". In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*. IEEE. 2001, pp. 279–285.