Dissolve: A Distributed SAT Solver based on Stålmarck's Method

Julien Henry[1], Aditya Thakur[2], Nicholas Kidd[3], and Thomas Reps[1,4]

[1] University of Wisconsin; Madison, WI, USA {jhenry,reps}@cs.wisc.edu
[2] Google, Inc.; Mountain View, CA, USA avt@google.com
[3] Google, Inc.; Madison, WI, USA nkidd@google.com
[4] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. Creating an effective parallel SAT solver is known to be a challenging task. At present, the most efficient implementations of parallel SAT solvers are *portfolio* solvers with some heuristics to share *learnt clauses*. In this paper, we propose a novel approach for solving SAT problems in parallel based on the combination of traditional CDCL with Stålmarck's method. In particular, we use a variant of Stålmarck's *Dilemma Rule* to partition the search space between solvers and merge their results.

The paper describes the design of a new distributed SAT solver, called Dissolve, and presents experiments that demonstrate the value of the Dilemma-rule-based approach. The experiments showed that running times decreased on average (geometric mean) by 25% and 17% for SAT and UNSAT examples, respectively.

1 Introduction

State-of-the-art solvers for the propositional satisfiability problem (SAT) perform very well and solve many large-size industrial problems, even though they tackle an NP-complete problem. One of the main reasons for this success is the *Conflict Driven Clause Learning* algorithm [22], which combines resolution-based clause learning, efficient heuristics for decisions, and aggressive restarting.

Parallel and/or distributed systems are widely used to improve software performance. With the availability of many-core computers, or the success of cloud computing, one can run software on a local machine in parallel with 64 cores or more, or distribute it on thousands of cores in the cloud. However, parallelizing SAT solving remains a very challenging task. At present, the best approach to parallel SAT solving is to execute of distinct sequential solvers in parallel with different settings and heuristic choices. Indeed, it commonly happens than just changing the random seed in a CDCL solver significantly changes the solving time of a given benchmark. These parallel or distributed solvers are referred to as *portfolio* solvers and, in their simplest form, do not exchange any information between their internal sequential solvers. Current best practice is to share some information by exchanging *learnt clauses* between the different sequential solvers. Recent research has focused on finding what the best heuristics are for sharing learnt clauses [9, 2, 24, 4].

An alternative to CDCL is the SAT-solving algorithm from Stålmarck[25]. This algorithm is a *validity checker*: it determines whether a propositional-logic formula is *valid* or not. A formula φ is valid if and only if its negation $\neg\varphi$ is unsatisfiable. Stålmarck's method has the advantage over CDCL of being intrinsically parallelizable. A key component is the Dilemma rule, which partitions the search space to solve different subproblems—thus allowing easy parallelism—and merges the information it learns from the different branches.

In this paper, we describe a new approach to massively parallel SAT solving, based on a combination of Stålmarck's method and sequential CDCL-based SAT solving. The intuition behind our work is the following: the Dilemma rule is a simple though efficient way of parallelizing SAT solvers, which also allows learnt clauses and other information obtained from different search spaces to be exchanged. To handle a formula φ, we use three phases:

- **Splitting:** The Dilemma rule splits the problem φ into smaller subproblems $\varphi_1, \ldots, \varphi_n$ such that $\varphi \equiv \bigvee_{1 \leq k \leq n} \varphi_k$. For instance, a simple way of splitting φ in two is to select an undecided variable and assume that it is true in one branch, and false in the other. Splitting creates n sub-problems that can be solved independently.
- **Solving:** The n created subproblems are processed in parallel using different sequential solvers. Solvers make progress in different directions by learning different learnt clauses. These learnt clauses are implied by φ, and thus also hold for the original problem φ.
- **Merging:** The n results are merged together to decide what new Dilemma-rule split to issue next. Clauses learnt from any branch are inserted into a clause database, along with extra information used for making heuristic choices.

The solving step takes advantage of a pool of SAT solvers running in parallel and processing the various SAT queries. Using the mechanism of solving with assumptions available in existing CDCL implementations, parts of the state of the solvers can be maintained across queries to speed up the computation.

Contributions. The contributions of the paper can be summarized as follows:

- We propose a novel parametric framework for parallel SAT solving, which combines the best of both Stålmarck's method and CDCL. In contrast to most existing approaches, our algorithm for parallel/distributed SAT solving shares not only learnt clauses, but also information to guide heuristic choices.
- We present heuristic and technical choices for some key features of the algorithm, in particular how to apply the Dilemma rule.
- We describe a new parallel/distributed SAT/SMT solver, called *Dissolve*, which implements our approach and runs on distributed machines. It is one of the only SAT/SMT solvers able to run on thousands of machines.
- We present extensive experimental results showing the importance of the key features of our framework. We show that the Dilemma-rule based approach improves the overall time performance of SAT instances by 25% and UNSAT instances by 17%, and solves more SAT instances than the *portfolio* approach.

Organization. The remainder of this paper is organized as follows: Section 2 gives an overview of Stålmarck's method. Section 3 describes our new algorithm for parallel/distributed SAT solving. Section 4 presents our implementation, *Dissolve*, as well as our experimental evaluation. Section 5 discusses related work. Section 6 draws some conclusions and presents future work.

2 Preliminaries: Stålmarck's Method

In this paper, we use $\mathbf{1}$ and $\mathbf{0}$ to denote the propositional constants *true* and *false*. A propositional variable or constant and its negation are both called *literals*. The equivalence of two literals a and b is denoted by $a \equiv b$. If a, b, c are in the same equivalence class, we write $[a, b, c]$. We use $\mathbf{0} \equiv \mathbf{1}$ to denote a contradictory equivalence relation.

2.1 Overview of Stålmarck's Method

We illustrate Stålmarck's method with the following example: consider the tautology $\varphi = (a \wedge (b \vee c)) \Leftrightarrow ((a \wedge b) \vee (a \wedge c))$. Stålmarck's method first decomposes this formula φ into a list of *integrity constraints*, by creating a fresh variable v_i for each subformula of φ:

$$\begin{array}{lll} v_1 \Leftrightarrow (v_2 \Leftrightarrow v_3) & v_2 \Leftrightarrow (a \wedge v_4) & v_3 \Leftrightarrow (v_5 \vee v_6) \\ v_4 \Leftrightarrow (b \vee c) & v_5 \Leftrightarrow (a \wedge b) & v_6 \Leftrightarrow (a \wedge c) \end{array}$$

In any assignment to the variables that satisfies the integrity constraints, the truth value of φ is given by v_1 . The principle of Stålmarck's method is to compute *formula relations*, i.e., implied assignments of variables to $\mathbf{1}$ or $\mathbf{0}$, until a relation $\mathbf{1} \equiv \mathbf{0}$ is derived, meaning that the formula is unsatisfiable. To prove that φ is a tautology, we start by assuming $v_1 \equiv \mathbf{0}$, and perform propagations with the propagation rules described in Figure 1 to obtain new equivalence relations.

$$\frac{p \Leftrightarrow (q \vee r) \qquad p \equiv \mathbf{0}}{q \equiv \mathbf{0} \qquad r \equiv \mathbf{0}} \ \textsc{Or1} \qquad\qquad \frac{p \Leftrightarrow (q \wedge r) \qquad q \equiv \mathbf{1} \qquad r \equiv \mathbf{1}}{p \equiv \mathbf{1}} \ \textsc{And1}$$

Fig. 1. Propagation rules

In general, these rules are not sufficient to lead to a contradiction. As a consequence, once there are no more propagations to apply, Stålmarck's method makes use of the Dilemma Rule, illustrated in Figure 2: a literal v_i is selected, and the problem is split into the two cases $v_i \equiv \mathbf{0}$ and $v_i \equiv \mathbf{1}$. Both branches are saturated with propagations and produce two distinct formula relations R_1 and R_2. Finally, the two cases are merged by intersecting the set of tuples present in both formula relations R_1' and R_2'. The reason why this step is correct is that if a relation holds both if $v_i \equiv \mathbf{0}$ and if $v_i \equiv \mathbf{1}$, then it also holds in general.

Applying the Dilemma Rule until the computed formula relation converges might not lead to a contradiction. It is also possible to use the Dilemma Rule recursively, which intuitively corresponds to selecting a set of k variables, and splitting into 2^k branches with all possible combination of assignments to the k variables.

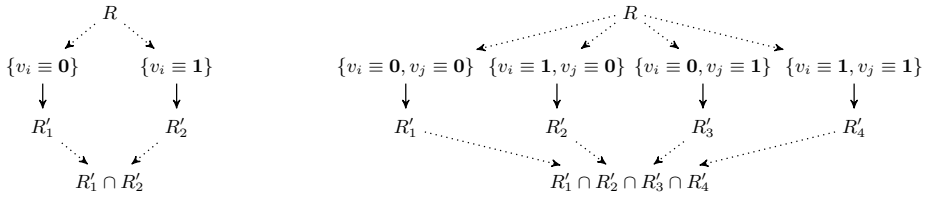

Fig. 2. Dilemma Rule, when splitting on one variable v_i (left) or two variables v_i, v_j (right).

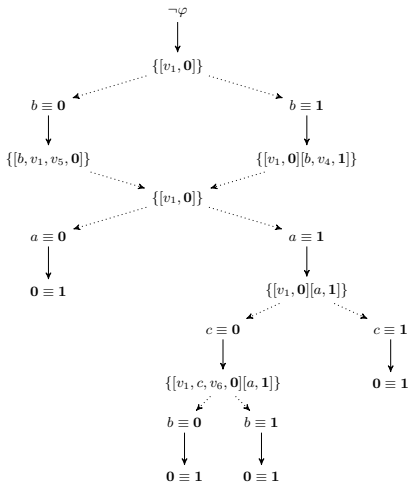

Fig. 3. Dilemma rule applied to our example proves that $\neg\varphi$ is unsatisfiable. Solid lines represent the computation of the transitive closure of the formula relation with the propagation rules from Figure 1. Dotted lines represent Dilemma-rule splits and merge.

3 Dilemma-Rule-based Parallel SAT-Solving

In this section, we present how to use an approach inspired by Stålmarck's method to implement a new parallel SAT algorithm. We first present an initial simplified version of our final algorithm.

3.1 Dilemma-rule-based splitting

We propose a novel approach for distributed SAT solving based on a combination of CDCL-based SAT solvers and Stålmarck's Dilemma rule.

While the Dilemma rule is a natural way of doing search-space partitioning, a pure Dilemma-rule based algorithm would require exchanging data too often between the different sequential solvers, namely, each time the application of the propagation rules converges. In a distributed setting, where the communication costs between solvers is important, one can reduce the communication costs by using CDCL-based sequential SAT solvers. Dilemma rules are applied in sequence until an answer is found (Algorithm 1). The algorithm is parameterized by an integer k, which controls how many variables are used for splitting.

When applied to k variables, `Dilemma` creates 2^k SAT queries, which are solved separately and in parallel by a collection of CDCL SAT solvers with a budget limit (Algorithm 2). After the budget is reached, results from all solvers are coalesced using *merge* function that we describe later, and a new Dilemma split is triggered to make further progress in the solving. The `Dilemma` function, used in Algorithm 1 and defined in Algorithm 2, returns *sat* if one branch returns *sat*, and returns *unsat* either if all branches return *unsat*, or one branch returns *unsat* with a conflict clause of size 0: a conflict clause of size 0 means that an *unsat* core is *true*, and then the formula is *unsat* even without the assumptions from the Dilemma rule.

Algorithm 1: Dissolve main procedure

```
Function Dissolve(φ, k):
  L ← ∅                                          /* set of learnt clause */
  dl ← ∅                                         /* set of decision literals */
  pol ← ∅                                        /* variable polarities */
  ans ← unknown
  round ← 1
  (w_i)_{1≤i≤2^k} ← SATsolvers()                 /* we start 2^k CDCL SAT-solvers */
  while ans = unknown do
    seed ← new random seed
    budget ← Budget(round)
    (ans, L, dl, pol) ← dilemma(φ, L, dl, pol, seed, budget)
    round ← round + 1
  return ans
```

The key point of this combination of CDCL and Stålmarck's method is that it allows an efficient way of distributing SAT by sharing valuable information. Sequential solvers can exchange both information to reduce the search space—i.e., learnt clauses—and to improve heuristic decisions—i.e., lists of decision literals to pick for decisions and variable polarities.

In Stålmarck's papers, the step of combining the results from different branches is described as an intersection: one only keeps information present in all branches. However, in our settings, it is possible to do better. Because the learnt clauses L in CDCL are clauses implied by the original problem φ, they are independent of the extra assumption A_i used in the solver at the time it was learnt. In other words, $(\varphi \wedge A_i) \Rightarrow L$, but more importantly $\varphi \Rightarrow L$. For this reason, instead of *intersecting* the sets of learnt clauses returned by all branches, one can use their *union*, which eliminates a larger portion of the search space. Furthermore, decision variables and polarities returned from the various branches can be heuristically combined to update the internal states of the sequential solvers at the start of the next round.

An optimization to the parallel Dilemma rule consists of looking at the conflict clause returned by *unsat* branches. Let $v_1, \ldots, v_n$ be the splitting variables

Algorithm 2: Dilemma-rule-based split

Function `dilemma`$(\varphi, L, dl, pol, seed, budget)$`:`

```
/* We have 2^k sequential solvers (w_i)_{1≤i≤2^k} waiting for jobs        */
parallel foreach i ∈ [1, 2^k] do
    A_i ← Assumption(i, dl)   /* A_i = i-th of the 2^k assumptions using dl */
    (ans_i, L_i, dl_i, pol_i) = w_i.Solve(φ, A_i, L, pol, seed, budget)
    if ans_i is sat then
        return (sat, ∅, ∅, ∅)
if all ans_i are unsat then
    return (unsat, ⊥, ∅, ∅)
(L, dl, pol) ← merge({(A_i, L_i, pol_i)}_{1≤i≤2^k})
return (unknown, L, dl, pol)
```

Function `Solve`$(\varphi, A_i, L, pol, seed, bgt)$`:`

```
/* use a standard CDCL solver, update its state with L, pol, seed, and
   solve the formula φ ∧ A_i with a timeout bgt. Return the result
   ans_i ∈ {unknown, sat, unsat}, and new clauses/literals/polarities
   L_i, dl_i, pol_i                                                      */
return (ans_i, L_i, dl_i, pol_i)
```

in the Dilemma rule. Suppose for example that a conflict clause for an *unsat* branch is $\neg v_1 \vee v_2$. Then, $v_1 \wedge \neg v_2$ is an unsatisfiable core of the formula. One can then interrupt all the other branches that assume $v_1 \wedge \neg v_2$.

While our contribution might seem very different from the original Stålmarck's method—a Dilemma-split runs for a much longer period of time, usually in the order of a few seconds to a minute, while Stålmarck's method does several mergings per second—, we argue that our algorithm explores the search space using the same approach to search-space exploration as Stålmarck's method:

- split the problem into subproblems
- learn "everything" you can about the subproblems that is relevant to the original instance (or at least "as much as you can")
- merge the knowledge from all instances, and restart with a new and different split.

3.2 Strategies and Heuristic Choices

The algorithms presented in this paper can be tuned in many ways with various strategies and heuristics. In the following, we present some important strategy choices that can significantly influence the efficiency of the implementation.

Distribution strategies

Suppose that we have 2^k sequential solvers available, either running on separate cores or on separate machines. The distribution strategy governs how SAT queries are spawned: Dilemma-rule-based splits can indeed be parameterized with the number of variables used when splitting:

– Full search-space partitioning: partition the search space with a Dilemma split using k variables $v_1, \ldots, v_k$. Each solver instance explores one of the 2^k partitions.
– Portfolio strategy: each solver instance uses a different random seed. Each of the 2^k solvers will solve one of the 2^k spawned SAT queries.
– Portfolio and search-space partitioning: use a combination of the above two strategies.

Solve-time strategy
The solver-time strategy governs the amount of time (or resources) each sequential solver instances has before it needs to report back to the main solver. It refers to the function *Budget* in Algorithm 1.

– Constant time: Each solver gets a constant time (say 10 seconds).
– Luby sequence: This strategy is borrowed from the restart strategies employed by current sequential SAT solvers. Let u be some unit time (say, 10 seconds). The time given for a call to a sequential solver in the ith iteration, t_i is $t_i = u * l_i$, where l_i is the ith digit in the Luby sequence defined by

$$l_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \\ l_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq i \leq 2^k - 1 \end{cases}$$

Example: the first digits of the Luby sequence are : 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,...

Instead of using execution time as a metric, one can also use another metric like the number of propagations, or the number of conflicts encountered before reporting back to the main solver.

Clauses returned
There are two possible strategies that govern what information is returned by each sequential solver when the result of the query is indeterminate.

– Length-restricted clauses: Return all clauses that have at most N literals.
– Clause ranking: Return the top m clauses based on an appropriate clause-ranking function (e.g., the *Literal Block Distance* (LBD) from [3]).
– Combination of the above: return the top m clauses of size $< N$ based on their LBD.

Dilemma-rule strategy
The choice of the k variables to use in the Dilemma rule from Stålmarck is done heuristically. There are several approaches that can be used to select these variables:

– Reuse the heuristic for decisions: given an integer k', when a sequential solver returns an answer, it also returns the next k' variables it would have chosen if it had to make a decision. The k' votes from all sequential solvers are counted and we select the k variables that receive the most votes to be the next Dilemma variables.
– Use a *distance* criterion: select the first variable using the idea above, then repeatedly select the next variable to be the one that occurs the most frequently within the same clauses as the variables already selected.

– Use the *community structure* of the *Variable Incidence Graph*. Nodes in this graph represent variables of the formula, and there is an edge between two variables for each clause that contains both variables. Recent work [1] has shown that community structures in this graph are a way of detecting useful learnt clauses. One could choose the k Dilemma variables from the same community, or from two connected communities. We did not use this approach because communities evolve over time with the learnt clauses, and are expensive to maintain.

3.3 Leveraging the Computational Resources

In practice, Algorithm 1 suffers from highly suboptimal usage of computational resources, with many CPUs being idle and waiting for a new query to solve. This imbalance is due to the fact that some queries are solved within milliseconds, while others run until timeout.

One possibility would be to let the sequential solvers run on the formula without Dilemma-rule assumptions instead of staying idle until the next query. The progress it makes during this time—e.g., learning new clauses—will be useful for the next query. However, we found that this approach often led processes to spend upwards of 90% of their CPU time running such non-Dilemma searches. This phenomenon made it impossible to perform an experimental study that separated the effects of the Dilemma-rule-based approach from a portfolio-based approach. In essence, such an approach degenerates to a portfolio-based search for a substantial proportion of the total computing resources available, and thus had similar performance on experiments.

In remainder of this subsection, we present a better adaptation of the previous algorithm that guarantees that CPU usage is close to 100% at all times, while still computing Dilemma splits almost all the time.

The algorithm uses a queue of Dilemma-rule-based SAT requests that are distributed to idle SAT solvers by a scheduler (Algorithm 3). The queue is filled by a SAT-queries producer (Algorithm 3), triggered when the queue becomes empty. The producer merely creates a new round of 2^k SAT queries from a new Dilemma-rule-based split. Once the round is issued, Algorithm 3 waits for the 2^k SAT replies and merges their results—e.g., their learnt clauses and other exchanged data—with the global state.

The new algorithm breaks the property of the algorithm in Section 3.1 of executing one Dilemma-split at a time, which implies that the learnt clauses, decision literals, and polarities given as input to `dilemma` differ from those that arise with the initial algorithm. When issuing a new round r, these learnt clauses, literals, and polarities are now based on the SAT replies received since the last round $r-1$. In practice, these SAT replies can come from any round less than or equal to $r-1$, because different Dilemma splits can be processed in parallel. As a consequence, if we denote by R the set of SAT replies that have been received since the start of the last Dilemma-split:

– the new polarity for a variable v is set to the most popular polarity value for v in R.

Algorithm 3: Dissolve with Asynchronous Dilemma-splits

```
1  Function scheduler():
2  |  Q ← empty                                          /* queue of SAT queries */
3  |  while True do
4  |  |  (φ, A_i, L, pol, seed, bgt, R) ← Q.pop()       /* blocking if Q is empty */
5  |  |  w ← getIdleWorker()                            /* blocking until a worker is idle */
6  |  |  Fork R.push(w.Solve (φ, A_i, L, pol, seed, bgt))

7  Function producer():
8  |  round ← 1
9  |  while True do
10 |  |  if size(Q) < number of idle workers then
11 |  |  |  seed ← new random seed
12 |  |  |  bgt ← Budget(round)
13 |  |  |  Fork
   |  |  |  |  result ← dilemma(round, φ, L, dl, pol, seed, bgt)
   |  |  |  |  if result ≠ unknown then
   |  |  |  |  |  Cancel all computation and return result
14 |  |  |  round ← round + 1

15 Function dilemma(round, φ, L, dl, pol, seed, bgt):
16 |  R ← new queue          /* SAT replies from round round will go there */
   |  /* Issue the requests                                                  */
17 |  parallel foreach i ∈ [1, k] do
18 |  |  A_i ← Assumption(i, dl)
19 |  |  Q.push(< φ, A_i, L, pol, seed, bgt, R >)
   |  /* Process the 2^k SAT solver replies                                  */
20 |  parallel foreach i ∈ [1, k] do
21 |  |  (ans_i, L_i, dl_i, pol_i) ← R.pop()             /* blocking if R is empty */
22 |  |  if ans_i = sat then
23 |  |  |  return sat
24 |  |  (L, dl, pol) ← merge((L, dl, pol), (L_i, dl_i, pol_i))
25 |  if ∀i ∈ [1, k], ans_i = unsat then
26 |  |  return unsat
27 |  return unknown

28 Function Solve(φ, A_i, L, pol, seed, bgt):
   |  /* same as in Algorithm 2                                             */
```

– the new decision literals are selected based on the decision literals received in R.

The strategy for exchanging learnt clauses in this case is presented in Section 3.4.

3.4 Exchanging Learnt Clauses

The efficiency of the approach depends significantly on the strategy for sharing learnt clauses across solvers. When returning an answer from a SAT query, each SAT solver also return a set of learnt clauses. Clauses from each solver then have

to be merged, and one has to decide which of the learnt clauses should be shared with all solvers in the next round.

UBTree data structure. Because clauses are sets of literals, an efficient way of representing a set of clauses is to use a UBTree (*Unlimited Branching Tree*) [14], an efficient structure to store this set of clauses while checking for clause subsumption to avoid redundant clauses and do some cheap simplifications.

A node in a UBTree has three components: 1. the element e it represents, in our case a literal; 2. its children, i.e., a set of other nodes; 3. a Boolean flag that tells whether the clause defined by the path from the root to this node is in the set.

The difference with [14] is that, in our case, it is possible to simplify the UBTree by removing the children of any node where the flag is set, because the longer clauses will be subsumed by the smaller one. Figure 3.4 shows an example of insertions into a UBTree with simple subsumption simplifications. Note that in our setting, the UBTrees are not fully simplified: the first insertion of (v_1, v_2, v_4) could lead to a simpler UBTree with only the clause (v_1, v_2, v_4) itself. It is then possible to further simplify the UBTree when inserting a clause C in the following way:

- Query for the set S of supersets of C,
- Remove all the nodes that are only used for clauses in S.

Because this last simplification technique involves more expensive searches, we did not use it in our experiments in Section 4.

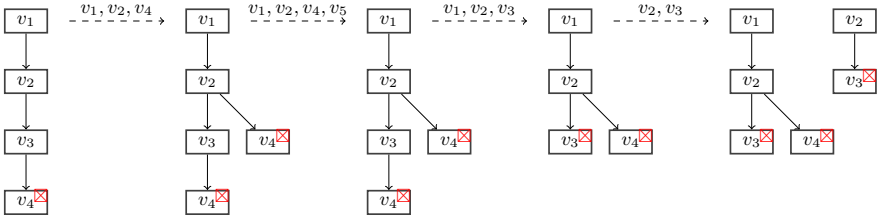

Fig. 4. Evolution of a UBTree, when inserting the clauses $\{v_1, v_2, v_4\}$, $\{v_1, v_2, v_4, v_5\}$, $\{v_1, v_2, v_3\}$, and $\{v_2, v_3\}$.

Clause-Sharing Policy. The fact that different Dilemma-rule-based splits can execute in parallel makes it complicated to design an efficient strategy to send learnt clauses to sequential solvers. In particular, it is sometimes the case that some of the sequential solvers will run a given request until timeout, while in the meantime other solvers will solve many requests that lead to fast *unsat* answers. Because queries are run on solver instances that retain information across queries that are assigned to them, one needs a strategy that avoids resending the same learnt clauses when issuing a new request, but also guarantees that high-quality learnt clauses are eventually sent to all sequential solvers. To do so, we maintain different sets of learnt clauses based on their quality and the time they have been returned by a solver. We index the various sets of clauses using a tuple (t, n) of integers. t is an index for the time frame from the start of the t-th Dilemma-rule split (also called round t) to the start of the t+1-th round. For a given time frame

indexed by t, we use N different UBTrees that we denote by $T_{t,1}, T_{t,2}, \ldots, T_{t,N}$. During the time frame defined by t, learnt clauses received from the various terminating SAT queries from all rounds are inserted into $T_{t,1}, T_{t,2}, \ldots, T_{t,N}$ based on some measure of quality: in practice, this measure is either the size of the clause or its LBD. The highest rated clauses are inserted in $T_{t,1}$, etc., and the less important clauses in $T_{t,N}$. In practice, we used $N = 3$.

When a solver returns the answer to a SAT query from round r, it is sent a new request from the latest round k, where $k \geq r$. At this time, it is also sent the "best" learnt clauses that it did not yet receive, in order of priority, until a fixed limit on the number of clauses is reached. Clauses in the highest-priority sets are sent first, i.e., $T_{r,1}$, then $T_{r+1,1}, \ldots, T_{k,1}$. If the limit is not reached, clauses are sent from the $(T_{i,2})_{r \leq i \leq k}$ sets, and so forth. In this way, a solver S is brought up to date with respect to learnt clauses obtained during rounds while S was busy on a round-r query.

Once all solvers are working on queries from rounds greater that r, the UBTrees from rounds less than or equal to r can be discarded. Also, a UBTree representation is only required for the latest round, while a textual CNF-format representation of the UBTree is sufficient for each previous round.

Example: Figure 5 gives an example of our clause-sharing policy. The latest round that has been started is 8. The first non fully completed round is 5, which means that all learnt clauses learnt before round 5 have already been deleted. **(1)** Solver1 returns from solving a query from round 6. **(2)** It is then sent clauses from rounds 6 and 7, in order of quality. **(3)** Solver2 returns from solving a query from round 5. **(4)** It is then sent clauses from rounds 5 through 7. UBTrees from round 8 are still under construction.

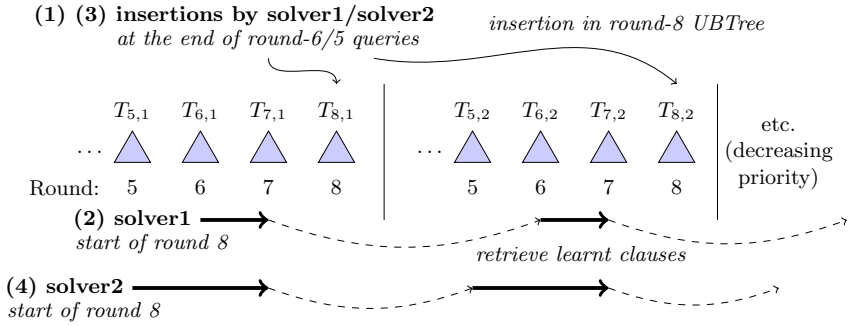

Fig. 5. Example of learnt-clause insertion and selection.

4 Experimental Evaluation

This section presents our experiments to evaluate the approaches used in Dissolve. They have been conducted on a 64-core Linux machine with AMD Opteron 6376 processors. We used the set of benchmarks from the 2015 SAT-race,[5] composed of 169 satisfiable and 117 unsatisfiable benchmarks of various difficul-

[5] http://baldur.iti.kit.edu/sat-race-2015/index.php

ties. The experiments were designed to answer the following questions: (i) Is the Dilemma rule a good criterion for splitting the problem for exploration on the different cores? (ii) How does clause-sharing perform when used with the Dilemma-rule and the *portfolio* approaches? (iii) How does Dissolve compare against other state-of-the-art multi-core solvers?

Dissolve is implemented in the Go language and features two kind of components: a *main solver* and a pool of *workers*. Each worker is connected to a dedicated SAT solver through a pipe interface, receives SAT queries from the main solver, and processes them with its solver. In theory, one could connect any available SAT solver, provided a small API is implemented—e.g., some import/export functions for learnt clauses, polarities, decision variables, etc. In our implementation, each worker's SAT solver is Glucose 3.0 [3], which is itself based on Minisat [8]. Workers do not communicate directly with each other; instead, all communication is with the *main solver*. The main solver is in charge of scheduling and sending the SAT queries to the workers, processing their results, and issuing new queries with the Dilemma rule. The main solver interacts with its workers using RPC calls and Google's protocol-buffer format to exchange data.

4.1 On the Importance of the Different Settings

The main purpose of our experiments is to establish whether the Dilemma-rule-based strategy can improve the efficiency of a SAT solver. We conducted experiments with *Dissolve*, in which we enabled/disabled some key features of the algorithm: exchange of learnt clauses, exchange of variable polarities, and Dilemma-rule/Portfolio strategy. In all cases, we have a pool of 2^k sequential solvers. By *portfolio* strategy, we mean the setting in which Dissolve rounds are made up of 2^k identical queries φ with a *different random seed* for each solver. In contrast, *Dilemma-rule-based strategies* are settings in which each Dissolve round splits the search space according to k Dilemma variables to produce 2^k different queries, which are solved with different random seeds.

Our earliest experiments showed that the Dilemma-rule-based strategy works best when the frequency of randomly picked decision literals is set to zero, while it should be a small value (around 5%) for the portfolio strategy. In other words, for the Dilemma-rule-based strategies, decision literals are only selected using an activity-based heuristic. In all the experiments that we report in this paper, Dilemma-rule-based settings use this 0% parameter, and portfolio-based settings use 5%. The other settings are equal.

At the beginning of each round, we send a maximum of 50000 learnt clauses to each of the 32 sequential solvers, in addition to the assumptions for a given solver. In our experiments, merging is performed when a sequential solver returns from a query with UNSAT or UNKNOWN. At that point, the solver also returns a list of 3125 ($= 2 * 50000/32$) learnt clauses that were not received or returned already. The first clauses in the list are unit clauses, followed by clauses of size 2, followed by other learnt clauses ordered with the `reduceDB_lt` ordering in Glucose (based on size and LBD), which gives a quality score. The first 100 of the latter clauses are flagged as "important."

1. All the learnt clauses of the same importance returned between the start of a round i and round $i+1$ are stored in the UBTrees $T_{i,1}$, $T_{i,2}$, or $T_{i,3}$.
2. During time-frame t, clauses of size <= 2 are stored in the UBTree $T_{t,1}$; "important" clauses are stored in UBTree $T_{t,2}$, and other clauses are stored in UBTree $T_{t,3}$.

Figure 6 shows cactus plots from our experiments on the 2015 SAT-race benchmarks set, separated into SAT (left) and UNSAT (right) instances. We used a budget of ≤20 seconds and ≤20,000,000 propagations for each query sent to a sequential solver. Our experiments were run with 32 solvers (i.e., $k = 5$). For each benchmark and each parameter setting, we use 3 runs and report the average solving time. The experiments show that exchanging learnt clauses between the sequential solvers gives huge benefits, especially for UNSAT benchmarks, with computation times becoming an order of magnitude faster than with the settings that do not share clauses.

The plots for the SAT benchmarks in Figure 6 show that the Dilemma-split-based settings have a significant advantage over the portfolio-based settings. On UNSAT instances, Dilemma splits have the advantage for most of the benchmarks, but are worse on the hardest 15% of the benchmarks. If we consider only the benchmarks solved in more than 10 seconds by both approaches, the geometric mean of the time ratios between "dilemma" and "portfolio" runs for each benchmark is 0.75 for SAT instances and 0.83 for UNSAT instances, which means that the Dilemma-rule-based approach significantly improves the overall performance in both SAT and UNSAT cases. Although they are not presented in this paper, similar experiments on the SAT-race 2013 benchmarks show similar results with Dissolve.

While the Dilemma rule seems less efficient for hard UNSAT instances, changing the time budget for each round to a smaller value gives better results, as depicted in Figure 7. A time limit of 5s instead of 20s is better on hard UNSAT cases but generally worse for SAT—it appears that 10 seconds is the sweet spot for these benchmarks.

4.2 Comparison with State-of-the-Art Tools

For the remainder of the paper, "*Dissolve*" refers to our implementation when used with the *Dilemma-rule*-based approach, exchanging learnt clauses and polarities (i.e., "dilemma" in Figure 6). Figure 8 presents an experimental comparison of Dissolve with some other state-of-the-art parallel SAT solvers. These results should be viewed with caution, because the different tools incorporate incomparable sets of features that make it hard to compare them on an equal footing. For instance, Dissolve is lacking features that could be added in the future, for instance, online simplification—Dissolve only relies on Minisat's simplification features as a preprocessing step; better approaches are described in [12]—or better heuristics to detect important learnt clauses. For instance, [4] describes a smarter approach to rating the importance of learnt clauses that we plan to integrate into Dissolve. Dissolve represents a substantially different point in the design space of SAT solvers from the other solvers used in the comparison. Dissolve is a distributed solver that is able to run in on a cloud-computing plat-

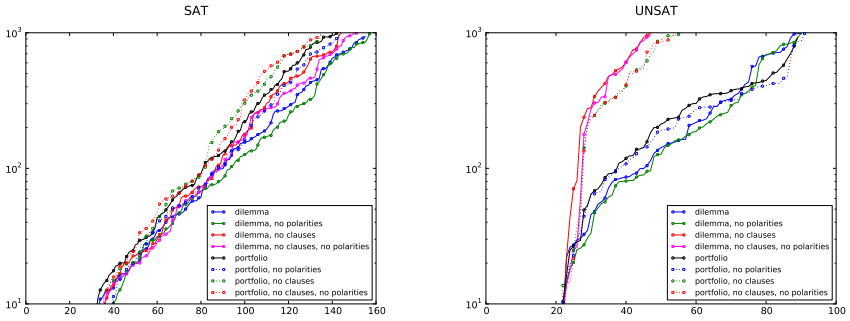

Fig. 6. Cactus plot of Dissolve runs with different parameters, on the SAT-race'15 benchmarks. *No clauses* (resp. *no polarities*) means that sharing learnt clauses (resp. polarities) is disabled. (It is recommended to view the diagrams in a PDF viewer so that they can be enlarged and viewed in color.)

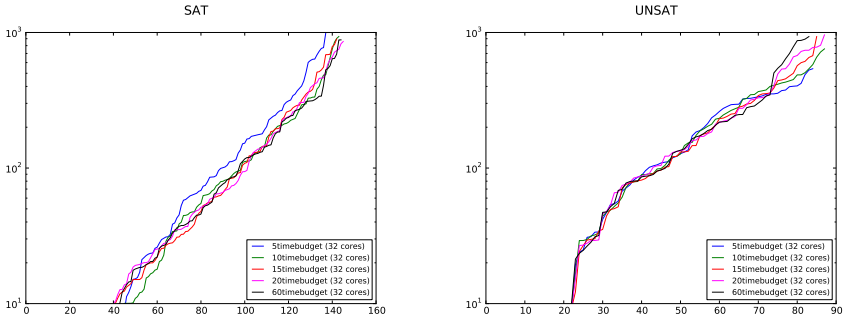

Fig. 7. Dilemma-splits approach, with different time budgets per round.

form. The other solvers are parallel solvers that work on a single machine, using multiple threads and shared memory, whereas Dissolve uses Google protocol-buffers to pass information between different solver engines. Because of communication overheads, Dissolve shares learnt clauses at a rate that is 2–3 orders of magnitude slower than the other solvers.

We can see that Dissolve performs *faster* in general compared to its competitors on SAT benchmarks, and is able to solve 154 benchmarks within the allocated time of 1000s, while Glucose-Syrup[4] solves 147 and plingeling[6] 144. It also obtains competitive results on UNSAT benchmarks. For these benchmarks, Glucose-Syrup is significantly more efficient, and solves 98 benchmarks while Dissolve solves 90, which suggests that one should incorporate the ideas from [4] with the Dilemma-rule-based approach.

5 Related Work

There has already been a substantial amount of work on parallelizing SAT solvers. Hamadi and Wintersteiger [11] put forth seven challenges for parallel SAT solving. In our work, we considered many of these challenges when designing our new algorithms, in particular what the authors refer to as *decomposition* and *knowledge sharing*.

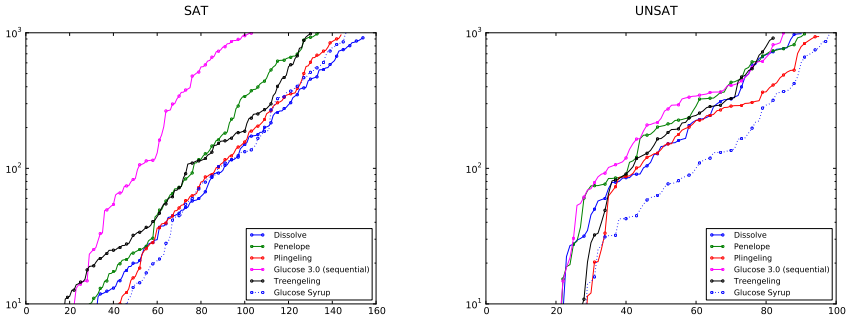

Fig. 8. Comparison between solvers, on 32 cores. Dissolve is the rightmost curve on SAT instances and solves more benchmarks; it is also competitive with the other solvers on UNSAT instances.

Katsirelos et al. [19] explore bottlenecks to parallelization of SAT solvers. Their findings suggest that efficient parallelization of SAT is not merely a matter of designing the right clause-sharing heuristic. They suggest exploring solvers based on other proof systems that produce more parallelizable refutation strategies. Our approach can be seen as an attempt to use such a more parallelizable strategy—namely, Stålmarck's method—while still reusing all the good engineering that has gone into existing CDCL solvers.

Even though most of the existing work has focused on portfolio approaches, there also exists some work that divides up the search space.

Earlier work from Hyvärinen [15, 17] uses *partition trees* to iteratively partition a formula. A partition function splits a formula into n sub-problems. [15, 17] propose different strategies to solve this partition tree, referred to as *plain partitioning* and *iterative partitioning*. Both approaches could be integrated into our framework with minimal work. However, in [17], clause sharing between sequential solvers is limited to unit clauses. Sharing information between solvers has also been proposed in earlier work from Hyvärinen et al. [16] and Manthey et al. [20, 21]. Sequential solvers only share those clauses that do not depend on a partitioning constraint. This approach requires a *flag-based* clause-tagging mechanism that we do not have to worry about in our approach. Hyvärinen et al. also proposed similar ideas for parallelizing SMT solvers [18].

Martins et al. [23] already propose an idea we use—namely, choosing the partition variables as a weighted sum of the decision variables returned by the sequential solvers. As an alternative to splitting the formula using partition variables, Hamadi et al. [10] describe a lazy-decomposition technique for distributing formulas, based on Craig interpolation.

Our work also has strong connections with Cube-and-Conquer [13], in the sense that we partition the problem using N variables. One of the key differences is that our partitions are only temporary, and are adapted over time. In contrast, Cube-and-Conquer spends a non trivial amount of effort at finding good partitioning variables first, before sending the resulting partitions to CDCL solvers. The benefits of our approach are that the splitting can be guided by the current search, and that the solver will be less sensitive to a given, po-

tentially bad, splitting-variable selection that would have been decided only at the beginning (e.g., a branch is still hard to solve). Another difference is that Cube-and-Conquer is based on a Lookahead heuristic to do the splitting, while ours is cheaper and based on a vote among all solvers based on their VSIDS information. When each sequential solver returns an answer to a query, it also returns a list of 10 variables by making 10 calls on the method pickBranchLit() from Glucose/Minisat (which is also used for decisions). The first pick is assigned a vote 10, the second 9, etc. We then merge all the scores from all sequential solvers, and the best 5 variables are picked for the next round. Other existing heuristics, including a lookahead heuristic, could be an interesting extension to our current implementation.

Some other work has focused on efficient heuristics to share learnt clauses. Audemard et al. [2] explore techniques for exchanging clauses within a parallel SAT solver that incorporates the LBD heuristic developed for Glucose. They incorporate these techniques in Penelope, a portfolio-based parallel SAT solver. More recent work by Audemard and Simon [4] reports excellent results for the clause-sharing method implemented in Glucose-Syrup. Hamadi et al. [9] described a dynamic control-based technique for guiding clause sharing.

There already exist a number of implementations of parallel SAT solvers, including those that we used for comparison with Dissolve in the experiments presented in Section 4.2. SatX10 [7] is a framework for implementing parallel SAT solvers based on the parallel execution language X10. HordeSAT [5] is a massively parallel portfolio solver designed to run on clusters on hundreds of cores. Comparing the efficiency of Dissolve with other distributed solvers on hundreds of cores has not been done yet, because of resource limitations, but is part of future work.

6 Conclusion

We described a novel approach for solving SAT problems on distributed machines by using a mixture of ideas from Stålmarck's method and state-of-the-art CDCL solvers. Successive Dilemma-rule-based splits are an efficient way to partition the problem into queries that explore different branches of the search space, while exchanging learnt clauses that hold for the original problem.

We presented a new tool, *Dissolve*, a distributed SAT solver that implements our approach. We showed experimentally that the Dilemma-splits approach improves on existing algorithms based on the portfolio strategy. In particular, we showed that *Dissolve* solves more SAT instances than competing state-of-the-art parallel solvers on the SAT-race 2015 benchmark set. The experiments also showed that the Dilemma-rule-based approach is a substantial improvement for both SAT and UNSAT benchmarks—improving SAT times by 25% and UNSAT times by 17% (computed using the geometric mean of the ratios of the "dilemma" and "portfolio" running times).

This motivates future work to further improve the efficiency of our implementation. One step towards improving *Dissolve*'s results would be to incorporate the clause ranking techniques used in Glucose-Syrup[4]. A second step would be to find better parameters, depending on the input formula to solve. Indeed,

as any other SAT solver, *Dissolve* has more than 40 tuning parameters, and it is very challenging to find which configuration is the best. We are investigating machine-learning techniques, in particular neural nets, to see whether it can help decide which parameter is best, given some features of the input formula. Also, we are interested in extending our approach and implementation to solve SMT problems. A prototype in which we replaced Glucose 3.0 with CVC4 to solve SMT formulas with *Dissolve* has been created, but at present it has not shown as good results as Dissolve does for Boolean SAT problems.

References

1. C. Ansótegui, J. Giráldez-Cru, J. Levy, and L. Simon. Using community structure to detect relevant learnt clauses. In *SAT*, 2015.
2. G. Audemard, B. Hoessen, S. Jabbour, J. Lagniez, and C. Piette. Revisiting clause exchange in parallel SAT solving. In *SAT*, 2012.
3. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Int. Joint Conf. on Art. Intell.*, 2009.
4. G. Audemard and L. Simon. Lazy clause exchange policy for parallel SAT solvers. In *SAT*, 2014.
5. T. Balyo, P. Sanders, and C. Sinz. Hordesat: A massively parallel portfolio SAT solver. In *SAT*, 2015.
6. A. Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In *Proc. SAT competition*, 2013.
7. B. Bloom, D. Grove, B. Herta, A. Sabharwal, H. Samulowitz, and V. A. Saraswat. SatX10: A scalable plug&play parallel SAT framework - (tool presentation). In *SAT*, 2012.
8. N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, 2003.
9. Y. Hamadi, S. Jabbour, and J. Sais. Control-based clause sharing in parallel SAT solving. In *Int. Joint Conf. on Art. Intell.*, 2009.
10. Y. Hamadi, J. Marques-Silva, and C. M. Wintersteiger. Lazy decomposition for distributed decision procedures. In *Int. Workshop on Parallel and Dist. Methods in verifiCation (PDMC)*, 2011.
11. Y. Hamadi and C. M. Wintersteiger. Seven challenges in parallel SAT solving. In *AAAI*, 2012.
12. M. Heule, M. Järvisalo, F. Lonsing, M. Seidl, and A. Biere. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res. (JAIR)*, 2015.
13. M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC*, 2011.
14. J. Hoffmann and J. Koehler. A new method to index and query sets. In *Int. Joint Conf. on Art. Intell.*, pages 462–467, 1999.
15. A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. Partitioning SAT instances for distributed solving. In *LPAR*, 2010.
16. A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In *CP*, 2011.
17. A. E. J. Hyvarinen and N. Manthey. Designing scalable parallel SAT solvers. In *SAT*, 2012.
18. A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina. Search-space partitioning for parallelizing SMT solvers. In *SAT*, 2015.
19. G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *AAAI*, 2013.

20. D. Lanti and N. Manthey. Sharing information in parallel search with search space partitioning. In *Learning and Intelligent Optimization*, pages 52–58, 2013.
21. N. Manthey, D. Lanti, and A. Irfan. Modern cooperative parallel SAT solving. In *Pragmatics of SAT*, 2013.
22. J. Marques Silva and K. Sakallah. GRASP – a new search algorithm for satisfiability. In *ICCAD*, 1996.
23. R. Martins, V. Manquinho, and I. Lynce. Improving search space splitting for parallel SAT solving. In *ICTAI*, 2010.
24. R. Martins, V. M. Manquinho, and I. Lynce. Clause sharing in parallel maxsat. In *Learning and Intelligent Optimization*, 2012.
25. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *FMSD*, 16(1):23–58, 2000.