SAT Simplification for Multi-core CPU Platforms

Popescu, Vlad-Andrei

Award date:
2021

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

SAT Simplification for Multi-core CPU Platforms

Master Thesis

Vlad-Andrei Popescu

Supervisors:
Anton J. Wijs
Muhammad Osama

Committee members:
Hans Zantema

version 1.2

Eindhoven, October 2021

Abstract. A multi-threaded SAT simplification algorithm presents an opportunity to accelerate solving compared to the state-of-the-art sequential simplification tools. While other parallel simplification tools exist, they either don't scale that well with the number of CPU cores, or in the case of the GPU simplifier SIGmA, need a dedicated hardware accelerator. A survey of simplification techniques is conducted in order to identify those that are effective in reducing the SAT formula and can be performed in parallel. A new multi-threaded algorithm combining several of these techniques is proposed and explained.

1 Introduction

The Boolean satisfiability problem (SAT) is NP-complete [7][16] and lies at the core of many techniques used in industry, most notably bounded model checking [5][3][23] and hardware equivalence [21][13]. As such, SAT solvers that are able to find solutions to such problems and do so quickly are desirable [9]. Two related techniques used to speed up the processing time of a SAT solver are preprocessing [19][18][9][22][1][2][11][14][12] and inprocessing [20][15]. In this paper, we collectively refer to them as SAT simplification.

A SAT problem is defined as determining if there exists a set of boolean variable assignments which satisfy a given boolean formula or not. The boolean formula is usually described by (or can be reduced to) the Conjunctive Normal Form (CNF), which is a conjunction of clauses. A clause is a disjunction of literals, which in turn are instances of boolean variables. SAT simplification works by reducing the number of variables and clauses in the CNF formula, reducing the work required by the SAT solver to find a solution. This is done either before the SAT solver is started (preprocessing) or intermittently during the execution of the solver (inprocessing).

Recently, a simplification tool that utilises GPU hardware for parallel processing was developed [19][18]. It utilises data independency in a novel way, achieving great results. Starting from this, we ask the following research question: Can a simplification algorithm that efficiently uses parallelization on multi-core CPUs be created? The goal of this paper is to answer this question, by describing and implementing a multi-threaded simplification algorithm that takes advantage of data parallelism in order to speed up existing simplification techniques. For this, several simplification techniques were surveyed and described, including the ways in which they can be run in parallel.

The structure of this thesis is as follows: firstly, the list of related work is introduced and summarized in Section 2. After that, preliminaries are defined in Section 3. Then, the survey of the simplification techniques is presented in Section 4. Next, the basis of the multi-threading techniques used are presented in Section 5, and the proposed multi-threaded algorithm is written and described in Section 6. After that, some details regarding the implementation of the program are described in Section 7. Finally, the benchmarking procedure is described and the results of the benchmarks are presented and discussed in Section 8.

2 Related Work

There are not many SAT simplification algorithms that use parallelism in literature. The most closely related work is that of K. Gebhart et. al. [10], where they describe and implement a fine-grained simplifier based on mutex locks. This strategy brings an improvement over a sequential approach, but the communication overhead and the reliance on mutex locks hurt scalability. In this paper, a more coarse-grained approach is proposed that tries to take advantage of data locality and minimizes the usage of mutex locks. While they cannot be completely avoided, in most cases a solution was found in which some of them can be bypassed (using delete flags, for instance).

M. Osama and A. Wijs presented the tool SIGmA [19][18], which describes a simplification algorithm that utilizes GPUs in order to speed up the computations. This is the starting point of this research, and the proposed algorithm tries to remedy the limitations of SIGmA. Not everyone (most notably in SAT competitions) has access to GPU hardware in order to accelerate SAT simplification, therefore an efficient CPU implementation does provide value to the SAT community. In general, this algorithm has proven to be quite efficient when compared to a sequential CPU preprocessor, but it is also worth exploring if the implementation of the same techniques in a multi-threaded CPU algorithm would result in better scaling, or what other possible benefits, regressions or trade-offs would such an algorithm bring.

The CaDiCaL solver developed by A. Biere [4] is a state-of-the-art SAT solver which contains a preprocessing stage. While the algorithm for both the preprocessing and the solving stages are sequential, this work is mentioned as the source-code is readily available and thus it provides a good baseline to benchmark against.

3 Preliminaries

In this section, domain specific terms and formulas are explained in order to fully understand the SAT problem, as well as the techniques that will be described later. We assume that the reader has knowledge of basic logic and reasoning.

- **SAT formula (F)** - The boolean formula representing the SAT problem. It is written in Conjunctive Normal Form (CNF). Recall that a CNF is a conjunction of clauses and each clause is a disjunction of literals. A formula is said to be satisfiable when all its clauses are satisfiable for the same variable assignment. It is usually written in two notations: The first one is the logic notation, in which the conjunction is explicitly marked. The second one is the set notation, in which the clauses are treated as elements of a set, which is the formula. Both instances of the same formula are illustrated below as an example.

$$F = (a \vee b \vee \bar{c}) \wedge (b \vee c \vee d \vee e) \wedge (\bar{a} \vee \bar{e}) \wedge (c \vee \bar{d} \vee f)$$
$$F = \{(a \vee b \vee \bar{c}), (b \vee c \vee d \vee e), (\bar{a} \vee \bar{e}), (c \vee \bar{d} \vee f)\}$$

- **Clause (C)** - An arbitrary disjunction in a formula. For example, $(b \vee c \vee d \vee e)$ is the second clause of the formula given above. A clause is satisfiable when there is a variable assignment such that the clause evaluates to $true$. Clauses are sometimes treated as sets of literals.

- **(Boolean) variable (x, y)** - Can be assigned only one value ($true$ or $false$) across the entire formula.

- **Literal (a, $\bar{a}$, l)** - An instance of a variable inside the formula, which is either positive (e.g. a) or negative (e.g. $\bar{a}$). In the example given above, the first literal of the fourth clause is c, and the last literal of the first clause is $\bar{c}$. Sometimes, the difference between a variable and its positive literal must be explicit. In that case, $var(a)$ and $lit(a)$ are used, representing respectively variable a, and literal a.

  Literals denoted by letters at the beginning of the alphabet are considered to be either positive or negative (e.g. a and $\bar{a}$), and the letters at the middle of the alphabet (e.g. ℓ, m, n) are used to abstract the polarity of a literal, and the complement of it is marked by boolean negation (e.g. ℓ and $\neg\ell$).

- **Unit Clause** - A clause containing only one literal (e.g. (ℓ)). It is special in the sense that it (and therefore the formula) can only be satisfied if the single literal is set to $true$.

- **Subformula (F_1, F_x, F_ℓ)** - A subset of a formula. If the subscript is a number, then it denotes an arbitrary subformula. If the subscript is a variable or a literal, then it implies that the subformula contains all and only clauses containing that variable or literal.

$$F_x = \{C \in F | x \in C \vee \neg x \in C\} \qquad F_\ell = \{C \in F | \ell \in C\}$$

- **Resolution** - Operation in which two clauses are combined into a new one, called the resolvent. If one clause contains a literal that is also present in the other clause, but in its opposite form, then resolution can be performed on the respective variable as follows. The literals of the variable are eliminated from both clauses, and the rest of the literals are joined together to form a new clause. This is commonly referred to as partial resolution.

$$C \otimes_\ell D = (C \cup D) \setminus \{\ell, \neg\ell\}$$

In the example below, resolution is performed on variable b ($\otimes_b$):

$$(a \vee b \vee \bar{c}) \otimes_b (\bar{b} \vee \bar{c} \vee d \vee e) = (a \vee \bar{c} \vee d \vee e)$$

Resolution is more commonly interpreted as an operation between two subformulas. In this case, partial resolution is performed on each Cartesian pair, yielding a set of resolvents. As a requirement, one of the subformulas must contain the same literal of the variable on which resolution is performed on in every clause, and the other subformula must contain the opposite literal, again, in every clause.

$$F_\ell \otimes_\ell F_{\neg\ell} = \{(C \cup D) \setminus \{\ell, \neg\ell\}) | C \in F_\ell \wedge D \in F_{\neg\ell}\}$$

4 Simplification Techniques

In this section, the survey of simplification techniques used later in the algorithm is presented. A formal definition for each technique is given, where F is the CNF formula, C and D are clauses, and ite is short for an 'if-then-else' statement (i.e. $ite(x, y, z) \Leftrightarrow x \Rightarrow y \wedge \neg x \Rightarrow z$). Following this is an example of the technique being used, and other discussions including effectiveness, opportunities for parallelization, and reasons for inclusion in the algorithm.

4.1 Tautology Elimination

A rather trivial approach that is not often explicitly described. M. Heule et. al. [12] presented the usefulness of the explicit use of the technique in conjunction with variable addition methods. In its basic form, it eliminates a clause if it contains two opposite literals, and is therefore tautological.

$$TE(F) = F \setminus \{C \in F | \exists \ell \in C : \neg\ell \in C\}$$

In the example below, the clause contains both literals of variable b, and can therefore be eliminated.

$$(a \vee \bar{b} \vee \bar{c} \vee b) \Leftrightarrow (a \vee \bar{c} \vee true) \Leftrightarrow true$$

4.2 Bounded Variable Elimination

As described in the work of S. Subbarayan et.al. [22], this method eliminates a variable by applying resolution on it, but only if the number of literals does not increase. Variations exist where the cutoff is defined by the number of clauses, instead of literals.

$$\begin{aligned} BVE(F) =& ite(\exists_x : |F_x| \geq |TE(F_{lit(x)} \otimes_x F_{lit(\neg x)})|, \\ & BVE((F \setminus F_x) \cup TE(F_{lit(x)} \otimes_x F_{lit(\neg x)})), F) \end{aligned}$$

In the example below, BVE is applied on variable a. Note that the number of resolvent clauses excluding tautologies is not higher than the number of clauses containing variable a in the original formula. This can be the case even when the positive and negative literal subformulas contain more than two clauses.

$$\begin{aligned} C =& \{(a \vee b \vee c), (a \vee \bar{d})\} \\ D =& \{(\bar{a} \vee \bar{c}), (\bar{a} \vee b \vee f)\} \\ C \otimes_a D =& \{(b \vee c \vee \bar{c}), (\bar{c} \vee \bar{d}), \\ & (b \vee c \vee f), (\bar{d} \vee b \vee f)\} \\ BVE(F, var(a)) =& \{(\bar{c} \vee \bar{d}), (b \vee c \vee f), (\bar{d} \vee b \vee f)\} \end{aligned}$$

This technique has proven to be quite efficient at simplifying the CNF formula in the past [22][15][4][19].

4.3 Boolean Constraint Propagation

Described by M. Davis et. al. [8], this technique is primarily used when solving using the DPLL algorithm, or other derivatives. It can also be used as a simplification method if unit clauses are present. For example, after the resolution step of BVE. It works by identifying unit clauses as constraints, as the variable contained within can only take one value in order to satisfy the formula. Therefore, the satisfied clauses (including the unit clause) can be removed from the formula and the negation of the constraint literal can be removed from the rest of the clauses.

$$\begin{aligned} BCP(F) = &ite(\exists C \in F : C = (\ell), \\ &BCP((F \setminus F_\ell \setminus F_{\neg\ell}) \cup \{D \setminus \{\neg\ell\} | D \in F_{\neg\ell}\}), F) \end{aligned}$$

In the example below, the first clause is a unit clause, so the variable b is constrained to $true$. As such, the second clause is satisfied and can be eliminated, and the third clause loses the literal $\bar{b}$ because it evaluates to $false$.

$$\{(b), (a \vee b), (\bar{b} \vee c \vee \bar{d})\} \Rightarrow \{(c \vee \bar{d})\}$$

4.4 Pure Literal Elimination

First described by M. Davis et. al. [8], and used as part of the DPLL search algorithm. It states that if a variable presents itself as either a positive or a negative literal, but without its negation, all clauses containing that literal can be satisfied by assigning the variable the corresponding value. Therefore, all clauses that contain it can be removed from the formula.

$$PLE(F) = ite(\exists \ell : F_{\neg\ell} = \emptyset, PLE(F \setminus F_\ell), F)$$

In the example below, literal a is pure. By assigning variable a to $true$ we can satisfy the first and second clauses and eliminate them.

$$\{(a \vee b), (a \vee \bar{c} \vee d), (b \vee c \vee \bar{d})\} \Rightarrow \{(b \vee c \vee \bar{d})\}$$

4.5 Subsumption Elimination

Described by N. Eén et. al. [9]. It states that a clause which contains all the literals of another clause is redundant and can be eliminated.

$$SE(F) = F \setminus \{C \in F | \exists D \in F : D \subset C\}$$

In example below, the first clause is subsumed by the second one, so it can be eliminated.

$$\{(a \vee b \vee \bar{c} \vee d), (a \vee \bar{c})\} \Rightarrow \{(a \vee \bar{c})\}$$

4.6 Self-Subsumption Elimination

Similarly described by N. Eén et. al. [9]. When a clause subsumes another one except for one literal, but the partially-subsumed clause contains the negation on that literal, partial resolution can be performed on that literal. The resolvent is added to the formula, and because it subsumes the previously partially-subsumed clause C that it originated from, that one is eliminated, effectively removing one literal from the formula.

$$\begin{aligned} SSE(F) = &ite(\exists_{C \neq D \in F} : \exists_{\ell \in C} : \neg\ell \in D \wedge D \setminus \{\neg\ell\} \subset C, \\ &SSE((F \setminus C) \cup (C \setminus \{\ell\})), F) \end{aligned}$$

4.7 Blocked Clause Elimination

M. Järvisalo et. al. [14] first described this procedure. A clause is blocked if it contains a blocked literal. A literal is called blocked if partial resolution on it only results in tautologies. Variants of this technique only remove the blocking literal, others remove the whole clause. In any case, the satisfiability of the formula is preserved, but the logical equivalence may not be.

$$BCE(F) = F \setminus \{C \in F | \exists_{\ell \in C} : TE(\{C\} \otimes_{var(\ell)} F_{\neg\ell}) = \emptyset\}$$

In the example below, the second clause is blocked by literal $\bar{a}$. By performing resolution on a with the third clause we get a tautology because of literals c and $\bar{c}$. The same can be said about resolution on a with the fourth clause because of literals $\bar{d}$ and d. Since those were the only possibilities for resolution of the second clause on variable a, literal $\bar{a}$ is blocked, and therefore the second clause is also blocked. For the sake of the example, we ignore the fact that we could rather perform BVE on variable a instead.

$$\{(\bar{a} \vee b), (\bar{a} \vee c \vee \bar{d}), (a \vee \bar{c}), (a \vee d \vee e)\} \Rightarrow^* \{(\bar{a} \vee b), (a \vee \bar{c}), (a \vee d \vee e)\}$$

4.8 Eager Redundancy Elimination

M. Osama et. al. [20] first proposed this method. It removes clauses in the formula which can also be obtained by performing partial resolution between 2 other clauses in the formula on a given variable.

$$ERE(F) = F \setminus \{C \in F | \exists_{x, C_1, C_2 \in F} : C_1 \otimes_x C_2 = C\}$$

In the example below, the first two clauses can be resolved on variable a. The resolvent is equivalent to clause three in the formula. Thus, according to ERE, the third clause can be removed.

$$\{(\bar{a} \vee b), (a \vee c \vee \bar{d}), (b \vee c \vee \bar{d}), (a \vee d \vee e)\} \Rightarrow \{(\bar{a} \vee b), (a \vee c \vee \bar{d}), (a \vee d \vee e)\}$$

4.9 Hidden Literal Addition

M. Heule et. al. [12] described this process as a variation on the elimination techniques based on tautology, subsumption and blocked clauses. Before testing for those redundancies, the tested clause is augmented with inferred information from binary clauses that share a literal with it. This makes it easier to discover redundancies in the clause and potentially eliminate it. If no redundancy was found however, the literal addition operations are reverted.

$$HLA(C, F) = C \cup \{\neg m | \exists D \in F \setminus \{C\} : \exists \ell \in C : D = (\ell \vee m)\}$$

In the example below, the second clause is a binary clause that shares literal b with the first clause. As such, we can use the second clause to augment the first one and add the negated remaining literal $\bar{d}$.

$$\{(a \vee b \vee \bar{c}), (b \vee d)\} \Rightarrow \{(a \vee b \vee \bar{c} \vee \bar{d}), (b \vee d)\}$$

This technique was chosen as an option to potentially enhance the clause elimination techniques (TE, SE, BCE).

4.10 Asymmetric Literal Addition

Similarly described by M. Heule et. al. [12], this method is a stronger version of HLA. Instead of adding information from binary clauses, all clauses aside from the tested one are considered for augmentation. For this to happen, the augmenting clause must share all but one literal. This technique has even greater potential to reveal redundancies than HLA [12]. Similarly, if the clause was not eliminated, the operation is reverted.

$$\begin{aligned} ALA(C, F) = & C \cup \{\neg m | \exists D \in F \setminus \{C\} : \exists \ell_1, \ell_2 \ldots \ell_n \in C : \\ & D = (\ell_1 \vee \ell_2 \vee \ldots \vee \ell_n \vee m)\} \end{aligned}$$

In the example below, the second clause shares all but one literal with the first clause. As such, the remaining literal d can be negated and added to the first clause.

$$\{(a \vee b \vee \bar{c}), (b \vee \bar{c} \vee d)\} \Rightarrow \{(a \vee b \vee \bar{c} \vee \bar{d}), (b \vee \bar{c} \vee d)\}$$

4.11 Hyper-Binary-Resolution

This method was first used as a preprocessor by F. Bacchus et. al [2]. It uses resolution between a clause and some binary clauses to generate a new binary clause. Using these new clauses, one can construct a more complete implication graph. Using the graph, equivalent literals can be reduced to one, failed literals can be used to deduce new unit clauses, and redundant edges can be removed from the graph, and thus the formula.

$$\begin{aligned} HypBinRes(F) =& ite(\exists C \in F : \exists x \in C : \forall y \in C \setminus \{x\} : \exists_z : (z, \neg y) \in F, \\ & HypBinRes(F \cup (x, z)), F) \end{aligned}$$

In the example below, let us consider the first clause. We observe that the negations of all but one literal can be found in binary clauses together with a constant literal $\bar{d}$. As such, hyper-resolution can be performed on the first, fifth and sixth clauses on variables a and c to produce the new binary clause, consisting of the remaining literal of the first clause and the constant literal of the fifth and sixth clauses.

$$\{(\bar{a}, c, \bar{e}), (a, b), (\bar{b}, c), (b, \bar{c}), (a, \bar{d}), (\bar{c}, \bar{d}), (\bar{c}, d)\} \Rightarrow (\bar{d}, \bar{e})$$

This technique was used as an inprocessor by itself in the past, with good results in problems with many binary clauses [1] [2] [11]. Including it as an option for this algorithm may be beneficial in tackling those kind of problems.

5 Multi-threading Techniques

In this section, the philosophy regarding the parallelization of the simplification techniques presented previously will be explained and justified. The main paradigm behind the design of the algorithm is that of data parallelism. According to this principle, worker threads can process independent data in parallel, achieving a speedup proportional to the number of workers when compared to sequential processing. In practice, there are some pieces of data that need to be shared between workers, and need to be protected. The method employed here is the usage of mutex locks.

In order to execute work in parallel, our algorithm makes use of a worker pool. The number of worker threads in the pool corresponds to the number of logical processors on the machine by default, but can also be manually configured. Work is assigned to the threads via the use of lambda functions. Once the work is done, the threads do not join the main program. They sleep in the worker pool, waiting to be called again.

Regarding the access pattern of our data structures, two major cases have been identified, requiring different strategies: Firstly, elements of a set (implemented as a vector) are (mostly) independent, and the set does not increase during execution. Secondly, elements of a queue (also implemented as a vector) are (mostly) independent, but new elements may be added to the queue during execution.

For the first case, the elements of the set are split and added to batches according to the number of workers and a preset maximum size of a batch. Then the worker threads are called to execute the necessary operation, each on a separate batch, until all have been processed. Mutex locks are only used if/when they are needed.

For the second case, the queue containing the elements must be accessed and modified inside the function to be called on all threads. Firstly, there is a mutex lock which protects the data in the queue when there is a pop or push operation. Since workers may push a new element at any time, the workers must not exit when the queue is empty, as long as other workers are running. As such, a condition variable waits inside the pop critical section, momentarily giving up the queue mutex. Other workers may notify a single waiting worker after they push a new element. The execution terminates when the queue is empty, and all workers are waiting inside the pop section.

Here are some notes regarding the specific implementation for some simplification techniques:

- **BVE** - The first strategy is used here. K. Gebhart et. al. [10] used a fine-grained approach using mutex locks on literals and clauses. M. Osama et. al. [19] suggested an election stage to identify and select independent variables first, then process them in parallel without the use of locks. The latter approach is used here.
- **BCP** - The second strategy is required, as new unit clauses can appear during execution. The 'delete later' flags can be set without locking by multiple threads without side-effects. When a literal is deleted from a clause, a mutex lock has to be used. PLE uses the same function, because the effect on the formula is the same.
- **SE** - Clauses being checked for subsumption can be processed in parallel without side-effects causing data racing [10]. If a race condition were to occur where $C \supset D$, and $D \supset E$, both C and D are candidates for removal. But since subsumption is transitive, it does not matter which one is removed first, since E subsumes both of them. As such, the first strategy can be used without mutexes.
- **BCE** - Checking for blocked clauses can be performed on each clause independently, because you cannot unblock a clause by removing another one from the formula. Therefore, the first strategy can be applied.
- **HLA** - Since some binary clauses can be used to augment a clause before removing it, a race condition could occur if the binary clauses are also candidates for removal. This is not the case here, because binary clauses are considered valuable. In a sense, the more binary clauses there are, the more the problem becomes similar to a 2-SAT problem, which is solvable in polynomial time [1]. As such, the first strategy can be applied here.
- **ALA** Unlike HLA, this technique can use any other clause for augmentation, not just binary ones. In order to guarantee that augmented clauses are not removed by other threads running in parallel, a *required* flag is set for those clauses. This flag acts as the counterpart of the *delete later* flag presented earlier, or in other words, a clause flagged with *required* is never deleted. This is a pessimistic approach, but it allows the first strategy to be applied without mutex locks.
- **Hyper-Resolution** - The second strategy is used when exploring the implication graph, by using multiple queues for the different operations. Mutex locks on literals (graph nodes) need to be used in order to safely update the ancestry data, and mark the exploration status of the nodes. Depending on the connectivity of the graph, the synchronisation overhead could be quite high.

6 Multi-threaded Algorithm

In this section, the simplification techniques presented earlier are combined into an algorithm. This algorithm thus developed tries to perform the stated simplification techniques in parallel as much and as efficiently as possible. The opportunities for multi-threading are also detailed here. The algorithm was structured into 4 stages, which will be discussed in order. This structure is presented in Algorithm 1. The starting point for this algorithm is represented by the work of Osama and A. Wijs [19], the structure of which can be recognised in Stage 4.

Algorithm 1 Simplify

```
1: function SIMPLIFY($F$)
2:     $OT \leftarrow \emptyset$
3:     $IG \leftarrow \emptyset$
4:     PREPARE($F, OT$)
5:     IMPLICATIONGRAPHREASONING($F, OT, IG$)
6:     CLAUSEELIMINATION($F, OT, IG$)
7:     BOUNDEDVARIABLEELIMINATION($F, OT, IG$)
8: end function
```

In short, the algorithm is structured as follows:

- Stage 1 - Preparation
  1. Initialization
  2. Boolean Constraint Propagation
  3. Pure Literal Elimination
- Stage 2 - Implication Graph Reasoning
  1. Explore Implication Graph
  2. Collapse equivalent literals
  3. Remove redundant edges
  4. Boolean Constraint Propagation (on failed literals and unit clauses)
  5. Hyper-Resolution (optional)
- Stage 3 - Clause Elimination
  1. Hidden Literal Addition (optional)
  2. Asymmetric Literal Addition (optional)
  3. Subsumption Elimination (Level 1 and 2)
  4. Tautology Elimination (Level 1)
  5. Blocked Clause Elimination (Level 2)
- Stage 4 - Variable Elimination
  1. Elect independent variables
  2. Bounded Variable Elimination

The reason for structuring Algorithm 1 like this is as follows. While initializing the data in Stage 1, unit clauses and pure literals can be found using only small additional checks. Moreover, removing these redundant literals and clauses early on may reduce the processing time needed for redundancy checking later.

Implication graph reasoning is applied next because eliminating equivalent variables reduces the number of candidates (and thus processing time) for BVE and may increase the chance for clause redundancies to be found. Hyper-resolution can also be performed while exploring the graph [11].

In Stage 3, all clause elimination techniques are grouped together because they have a similar parallel structure. They are grouped on two "levels" as BCE is stronger than TE [12]. Literal addition techniques are also included here to hopefully increase the number of redundant clauses found [12]. Clause elimination is done before BVE in order to reduce the number of clauses on which resolution can be performed and potentially reduce the number of resolvents to the point where more variables pass the heuristic test of BVE.

Finally, BVE is performed separately as its parallel structure does not allow an easy integration with other simplification techniques, other than checking resolvent clauses for redundancies.

6.1 Stage 1 - Preparation

In this stage (Algorithm 2), the input formula F is first stored in a data structure suitable for simplification operations. The clauses are represented by vectors of literals, and the formula is implemented as a vector of references to the clauses. Having clauses be represented by red-black trees was experimented with, but the penalty in access time was not worth it compared to the benefits from the insertion and deletion operations. Then, the Occurrence Table (OT) is initialized/reset in lines 4-6. Its role is to keep track of the clauses in which the literals appear. The structure is an array of vectors (occurrence lists) containing clause references. The OT allows for a quick way to determine all the clauses in which a certain literal is used.

The loop at line 7 processes the clauses in parallel. The OT is updated accordingly, and unit clauses are detected, the constraint literals being added in a queue. Operations on the OT are made atomic with the use of a mutex lock for each occurrence list. The constraint literal queue is also protected by another mutex. Then, the literals are checked in parallel (line 15) and the pure literals are also added in the constrained literal queue. Lastly, literals marked as constraints or pure are processed by the method which performs BCP and PLE.

Algorithm 2 Stage 1 - Preparation

```
 1: function PREPARE($F, OT$)
 2:     $F \leftarrow convert(F)$
 3:     $UnitOrPure \leftarrow \emptyset$
 4:     for $x \in lits(F)$ do                                   ▷ In parallel over $x$
 5:         $OT[x] \leftarrow \emptyset$
 6:     end for
 7:     for $c \in F$ do                                         ▷ In parallel over $c$
 8:         for $x \in c$ do
 9:             $OT[x].insert(index(c))$                         ▷ Atomic
10:         end for
11:         if $size(c) = 1$ then
12:             $UnitOrPure \leftarrow UnitOrPure \cup \{x\}$    ▷ Atomic
13:         end if
14:     end for
15:     for $x \in lits(F)$ do                                   ▷ In parallel over $x$
16:         if $OT[x] = \emptyset$ then
17:             $UnitOrPure \leftarrow UnitOrPure \cup \{\neg x\}$   ▷ Atomic
18:         end if
19:     end for
20:     BCP($UnitOrPure$, null, $F, OT$)
21: end function
```

The method performing BCP and PLE (Algorithm 3) is based on the procedure described by Davis et. al. [8] with some modifications to accommodate for parallel processing. Here, U represents a queue of literals. The worker threads each pop a literal (line 3) to process. This operation is made atomic by using a mutex lock and a condition variable.

Instead of removing the clauses containing the boolean constraint or the pure literal from the data structure, a flag is set instead (line 5). This way, race conditions are avoided without locking. When removing literals (line 9), operations are made atomic using a mutex lock for each clause to guarantee thread safety. If new unit clauses are found, the corresponding literals are pushed back in the queue (line 11). This operation is made atomic by locking the same mutex that is used when popping the literals from the queue. New binary clauses are also detected and added to a vector B (line 13). This vector is protected by another mutex lock. During this stage, the method is called with $B \leftarrow$ `null`, but binary clause detection will become useful in the next stage.

Algorithm 3 Boolean Constant Propagation + Pure Literal Elimination

```
function BCP($U, B, F, OT$)
    repeat                                        ▷ In parallel with worker pool
        $x \leftarrow U.pop()$                               ▷ Atomic
        for $c \in OT[x]$ do
            $c.deleteLater \leftarrow true$
        end for
        for $c \in OT[\neg x]$ do
            if $\neg c.deleteLater$ then
                $c.erase(\neg x)$                            ▷ Atomic
                if $c.size() = 1$ then
                    $U \leftarrow U \cup c[0]$               ▷ Atomic
                else if $B \neq$ null $\wedge\, c.size() = 2$ then
                    $B \leftarrow B \cup c$                  ▷ Atomic
                end if
            end if
        end for
        $OT[x].clear()$
        $OT[\neg x].clear()$
    until $U = \emptyset$
end function
```

6.2 Stage 2 - Implication Graph Reasoning

In this stage (Algorithm 4), the data in the implication graph is being processed. Here, P represents a queue of literals to be propagated, R represents a queue of literals of which the corresponding node in the graph should be reset, and E represents a queue of literals whose node should be explored. All these queues are protected by a mutex lock for each one of them. Also, there is the IG, which represents the implication graph. The role of the IG is to keep track of implication relations inferred by binary clauses. It consists of an array of references to node structures. A node structure contains a vector of references to parent nodes, another one for children nodes, and yet another vector for descendants that are not children. The IG is initialized using the method on line 7.

Algorithm 4 Stage 2 - Implication Graph Reasoning

```
function ImplicationGraphReasoning($F, OT, IG$)
    $P \leftarrow$ unit clauses in $F$
    $B \leftarrow \emptyset$
    $R \leftarrow \emptyset$
    $E \leftarrow \emptyset$
    $IG \leftarrow \emptyset$
    CreateIG($F, F, IG$)
    repeat
        BCP($P, B, F, OT$)
        CreateIG($B, F, IG$)
        $P \leftarrow \emptyset$
        ELR($R, F, IG$)
        for $l \in R$ do                                                    ▷ In parallel over $l$
            if $IG[l].status$ = explored then
                $IG[l].status \leftarrow$ unexplored
                $R \leftarrow R \cup IG[l].parents$                                ▷ Atomic
            end if
        end for
        for $l \in F$ do                                                    ▷ In parallel over $l$
            if $\forall_{x \in IG[l].children} IG[x].status = explored$ then
                $E \leftarrow E \cup \{l\}$                                        ▷ Atomic
            end if
        end for
        for $l \in E$ do                                                    ▷ In parallel over $l$
            if $\exists_{x \in IG[l].children} IG[x].status = unexplored$ then
                continue
            end if
            $IG[l].descendants \leftarrow IG[l].descendants \bigcup_{x \in IG[l].children} IG[x].children \cup IG[x].descendants$
            for $x \in IG[l].children$ do
                if $x \in IG[l].descendants$ then
                    $F \leftarrow F \setminus \{(\bar{l}, x)\}$                      ▷ Atomic
                end if
            end for
            if $\bar{l} \in IG[l].descendants$ then
                $P \leftarrow P \cup \{\bar{l}\} \cup IG[\bar{l}].children \cup IG[\bar{l}].descendants$     ▷ Atomic
            end if
            $G \leftarrow$ HyperRes($l, F, OT$)
            if $G \neq \emptyset$ then
                for $x \in G$ do
                    $F \leftarrow F \cup \{(\bar{l}, x)\}$                          ▷ Atomic
                    $IG \leftarrow IG \cup$ CreateIG($\{(\bar{l}, x)\}, F, IG$)
                end for
                continue
            end if
            $IG[l].status \leftarrow$ explored                                 ▷ Atomic
            $E \leftarrow E \cup IG[l].parents$                                ▷ Atomic
        end for
    until $P = \emptyset \wedge E = \emptyset$
end function
```

The role of the CreateIG method is to create or update the IG using the binary clauses in subformula U. During initialization, this method is called with $U \leftarrow F$. The operations are made atomic by a mutex lock for each graph node.

Algorithm 5 Create Implication Graph

```
1:  function CreateIG($U, IG, F$)
2:      for $c \in U$ do                                                   ▷ In parallel over $c$
3:          if $|c| = 2$ then
4:              $IG[\overline{c[0]}].children = IG[\overline{c[0]}].children \cup c[1]$       ▷ Atomic
5:              $IG[\overline{c[1]}].children = IG[\overline{c[1]}].children \cup c[0]$       ▷ Atomic
6:              $IG[c[0]].parents = IG[c[0]].parents \cup \overline{c[1]}$                    ▷ Atomic
7:              $IG[c[1]].parents = IG[c[1]].parents \cup \overline{c[0]}$                    ▷ Atomic
8:          end if
9:      end for
10: end function
```

Getting back to the main algorithm of this stage, the boolean constraints are propagated, and any new binary clauses are added to the IG. Then, the graph is scanned for Strongly Connected Components (SSCs) on line 12, and the cycles in the graph (or in other words, the equivalent literals) are reduced (see Algorithm 6). It uses the offline UFSCC algorithm developed by V. Bloemen et. al. [6] for cycle detection (line 3). Details for this algorithm are not provided in this paper, but in summary, it partitions the graph into SCCs (a single node can be considered its own SCC) in parallel using the OpenMP library [17], and it returns the representative node of each literal. One SCC has the same representative node which represents the node to which the SCC can be reduced to. Using this information, each literal is replaced by its representative literal if they differ (lines 7-12). The clause data is kept safe from conflicts by using a mutex lock for each clause. The representative literal for each cycle found is then pushed to the R queue to be marked as unexplored (line 11), along with any ancestor nodes found during exploration (inferred from the descendant data, as the graph is symmetrical). The operation is made atomic by using a mutex lock.

Algorithm 6 Equivalent Literal Reduction

```
1:  function ELR($R, F, IG$)
2:      repeat
3:          $reps \leftarrow$ UFSCC($IG$)
4:          for $v \in vars(F)$ do                                         ▷ In parallel over $v$
5:              $l \leftarrow lit(v)$
6:              $r \leftarrow reps[l]$
7:              if $l \neq r$ then
8:                  ReplaceLit($l, r, F$)
9:                  ReplaceLit($\bar{l}, \bar{r}, F$)
10:                 Remove $l$ and $\bar{l}$ from $OT$ and $IG$
11:                 $R \leftarrow R \cup \{r, \bar{r}\}$                        ▷ Atomic
12:             end if
13:         end for
14:     until $IG$ has no cycles
15: end function
```

Next in the main algorithm of the stage, all nodes that have all their children explored are included in the explore queue on line 21. This includes nodes without children. Then, the explore queue is processed. The exploration only starts if all children were previously explored (check made on lines 25-27). First, the indirect descendants (excludes direct children) are gathered on line 28. Then, the redundant edges are removed on line 31. After that, the literal is checked for failure. If the check passes, the negation of the literal, along with its children (and descendants, if the node has been explored) are added to the propagation queue (line 35), and the exploration continues with the next node in the exploration queue. Next, hyper-resolution is performed (see Algorithm 7).

Algorithm 7 Hyper-Resolution

```
1: function HyperRes($n, F, OT$)
2:     $F' \leftarrow copy(F)$
3:     $OT' \leftarrow copy(OT)$
4:     $U \leftarrow n.children \cup n.descendants$
5:     $U' \leftarrow U$
6:     BCP($U', F', OT'$)
7:     return $U' \setminus U$
8: end function
```

The method for hyper-resolution (Algorithm 7) is inspired by the work of Gershman et. al. [11]. Firstly, the transitive closure is computed on line 4. Then, the unit propagation closure is computed for the current node (line 6). Then only the nodes that are in the unit propagation closure, but not in the binary transitive closure (graph descendants) are returned. Note that instead of copying the whole formula and the OT, only a difference to the formula is calculated in order to keep the memory usage from increasing linearly with the number of workers, since nodes are explored independently, in parallel. The drawback is more processing time required.

Back to the main algorithm of the stage, if new edges have been found during hyper-resolution, the new clauses are added to the formula, the IG is updated, and the exploration terminates early (lines 38-44). Otherwise, the current node is marked as explored (line 45) and its parents are added to the exploration queue (line 46). The whole stage (except the initialization) repeats itself until both the propagation and the exploration queues are empty.

Please note that there are some differences between what is presented here and the actual implementation. The mutexes used are read/write mutexes for protecting the IG nodes, and exclusive mutexes for the queues. Secondly, there were issues caused by the part that removes redundant edges, causing the satisfiability of the formula to not be preserved. This was caused either by a flaw in the algorithm regarding what makes an edge redundant, or a faulty implementation. Lastly, a cap has been put on the number of times Hyper-Resolution can reset the stage, and it does not usually run until fixed point. This was done because the processing time for this part was quite large, and the unbound resets made it even worse.

6.3 Stage 3 - Clause Elimination

During this stage (Algorithm 8), several clause elimination methods are performed. There are two configurations: level 1 performs SE and TE, and level 2 performs SE and BCE. Moreover, each level can be further configured to augment the clauses through HLA or ALA before trying to eliminate them. Figure 1 illustrates this, where an arrow indicates that a method pointed from will reduce at least the same clauses as the method being pointed at (i.e. is ’stronger-than’). If ALA is performed, then SE is skipped. At the end, the OT is reduced (Algorithm 15). The IG does not need updating because binary clauses are not subject to elimination.

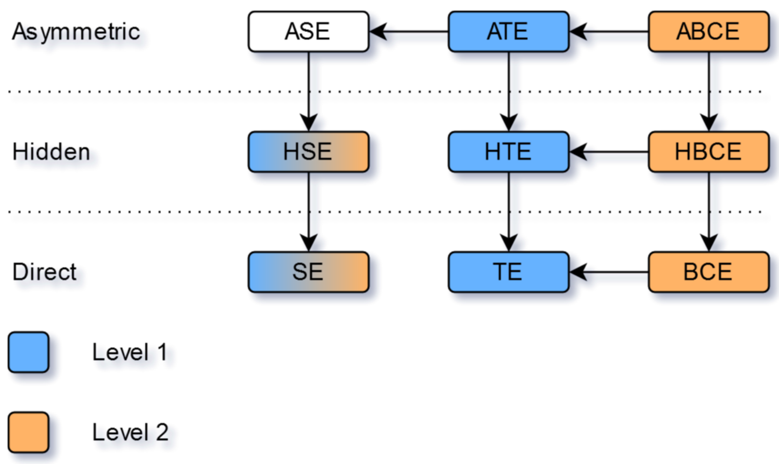

Figure 1: Clause elimination options. Arrows indicate a ’stronger-than’ relation.

Algorithm 8 Stage 3 - Clause Elimination

```
1:  function ClauseElimination($F, OT, IG$)
2:      for $c \in F$ do                                        ▷ In parallel over $c$
3:          if $c.size() > 2$ then
4:              $c' \leftarrow copy(c)$
5:              if CE($c', F, OT, IG$) then
6:                  $c.deleteLater \leftarrow true$
7:              end if
8:          end if
9:      end for
10:     $D \leftarrow \{d \in F | d.deleteLater \wedge \neg d.required\}$
11:     $F \leftarrow F \setminus D$
12:     if $literal\_addition = 2$ then
13:         for $d \in F$ do                                    ▷ In parallel over $d$
14:             $d.deleteLater \leftarrow false$
15:             $d.required \leftarrow false$
16:         end for
17:     end if
18:     ReduceOT($F, OT$)
19: end function
```

Both level 1 and level 2 configurations without literal addition can be done in parallel, because these procedures used were proven to be confluent [12]. The formula can be split in multiple batches and assigned to workers that process one batch at a time. HLA does not affect the parallelism, because binary clauses are excluded from the elimination candidates. For ALA, clauses that were used for augmentation, are marked as required, and cannot be deleted.

Algorithm 9 Clause Elimination

```
function CE($c, F, OT, IG$)
    if $c.required$ then
        return $false$
    else
        $R \leftarrow \emptyset$
        if $literal\_addition = 1$ then
            HLA($c, IG$)
        else if $literal\_addition = 2$ then
            $R \leftarrow$ ALA($c, F, OT$)
        end if
        $redundant \leftarrow false$
        if $literal\_addition < 2$ then
            $redundant \leftarrow$ SE($c, F, OT$)
        end if
        if $\neg redundant \land clause\_elimination = 1$ then
            $redundant \leftarrow$ TE($c, F, OT$)
        end if
        if $\neg redundant \land clause\_elimination = 2$ then
            $redundant \leftarrow$ BCE($c, F, OT$)
        end if
        if $redundant$ then
            for $c \in R$ do
                $c.required \leftarrow true$
            end for
            return $true$
        else
            return $false$
        end if
    end if
end function
```

Algorithm 10 Hidden Literal Addition

```
function HLA($c, IG$)
    repeat
        $c' \leftarrow copy(c)$
        for $x \in c'$ do
            for $y \in IG[\neg x]$ do
                $c \leftarrow c \cup \neg y$
            end for
        end for
    until $c = c'$
end function
```

Algorithm 11 Asymmetric Literal Addition

```
 1: function ALA($c, F, OT$)
 2:     $R \leftarrow \emptyset$
 3:     repeat
 4:         $c' \leftarrow copy(c)$
 5:         $D \leftarrow \emptyset$
 6:         for $x \in c'$ do
 7:             $D \leftarrow D \cup OT[x]$
 8:         end for
 9:         for $d \in D$ do
10:             if $\neg d.deleteLater$ then
11:                 $d' \leftarrow d \setminus c'$
12:                 if $|d'| = 1$ then
13:                     $c \leftarrow c \cup \neg d'[0]$
14:                 end if
15:                 $R \leftarrow R \cup d$
16:             end if
17:         end for
18:     until $c = c'$
19:     return $R$
20: end function
```

Both literal addition methods (Algorithm 10 and Algorithm 11) are fairly straight-forward. For HLA, the implication graph is used because we are only interested in the binary clauses. For ALA, a list of clauses used for augmentation is also returned in order to prevent them from being removed.

Algorithm 12 Subsumption Elimination

```
 1: function SE($c, F, OT$)
 2:     $D \leftarrow \emptyset$
 3:     for $x \in c$ do
 4:         $D \leftarrow D \cup OT[x]$
 5:     end for
 6:     for $d \in D$ do
 7:         if $d \subset c$ then
 8:             return $true$
 9:         end if
10:     end for
11:     return $false$
12: end function
```

Algorithm 13 Tautology Elimination

```
1: function TE($c$)
2:     for $x \in c$ do
3:         if $\neg x \in c$ then
4:             return $true$
5:         end if
6:     end for
7:     return $false$
8: end function
```

Algorithm 14 Blocked Clause Elimination

```
function BCE($c, F, OT$)
    for $x \in c$ do
        if Resolution($c, F[OT[\neg x]]) = \emptyset$ then
            return $true$
        end if
    end for
    return $false$
end function
```

The clause elimination methods (Algorithm 12, Algorithm 13, and Algorithm 14) are also straightforward. They return *true* in case they found the checked clause redundant, and *false* otherwise.

The method for reducing the occurrence table (Algorithm 15) is also fairly simple. It goes over the literals in parallel, and removes the clauses from the OT if they were marked to be deleted. As usual, the entries of the occurrence table are protected by atomic operations which use mutex locks. Note that the clauses are not deleted from the formula.

Algorithm 15 Reduce Occurrence Table

```
function ReduceOT($F, OT$)
    for $l \in OT$ do                                  ▷ In parallel over $l$
        for $c \in OT[l]$ do
            if $c.deleteLater = true$ then
                $OT[l] \leftarrow OT[l] \setminus c$
            end if
        end for
    end for
end function
```

However, preliminary tests showed that this stage takes a lot of processing time. This could be due to the large number of clauses that are candidates for subsuming the given clause. For this reason, a new approach was developed, called Reverse Subsumption Elimination (RSE). Here, instead of checking whether a clause can be subsumed by another clause in a subformula, it checks which clauses in a subformula can subsume a given clause (See Algorithm 16). This approach could be faster, but it does not support ALA. In the end, preliminary testing showed that this approach gives around the same level of performance as the initial one, some formulas showing a benefit, others, a deficit.

Algorithm 16 Reverse Subsumption Elimination

```
 1: function RSE($c, F, OT, IG$)
 2:     $Subsumed \leftarrow \emptyset$
 3:     for $l \in c$ do
 4:         $Candidates \leftarrow \emptyset$
 5:         if HLA is enabled then
 6:             for $l_{aug} \in IG[l].descendants$ do
 7:                 $Candidates \leftarrow Candidates \cup (OT[l_{aug}] \cap Subsumed)$
 8:             end for
 9:         else
10:             $Candidates \leftarrow OT[l]$
11:         end if
12:         if $l$ is the first literal of $c$ then
13:             $Subsumed \leftarrow Candidates$
14:         else
15:             $Subsumed \leftarrow Subsumed \cap Candidates$
16:         end if
17:         if $Subsumed = \emptyset$ then
18:             break
19:         end if
20:     end for
21: end function
```

Due to the poor performance observed on both these approaches, HLA and ALA ended up being only partially implemented, and not implemented respectively.

6.4 Stage 4 - Variable Elimination

In the last stage, BVE is performed in parallel (Algorithm 17). For this to be possible, independent variables must be elected for elimination. This is done using the LCVE method.

Algorithm 17 Stage 4 - Bounded Variable Elimination

```
 1: function BoundedVariableElimination($F, OT$)
 2:     $E \leftarrow$ LCVE( )
 3:     for $x \in E$ do                                   ▷ In parallel over $x$
 4:         HSE($x, F, OT$)
 5:     end for
 6:     for $x \in E$ do                                   ▷ In parallel over $x$
 7:         BVE($x, F, OT$)
 8:     end for
 9:     for $x \in E$ do                                   ▷ In parallel over $x$
10:         BCE($x, F, OT$)
11:     end for
12: end function
```

The Least Constrained Variable Election (LCVE) procedure is taken from the SIGmA algorithm [19]. According to this procedure, a number of variables are chosen from a pool of variables (usually all the variables in the formula) such that their literals do not share any clause they appear in. The chosen variables are called *elected*, and the variables no longer eligible to be elected are called *frozen*.

LCVE is presented in Algorithm 18 and works as follows: Firstly, a score is calculated for each variable in lines 3-13. The score is represented by the product between the number of occurrences of the positive and negative literal of the variable, or the maximum value of the two in case either one is zero. This heuristic is used in order to elect the variables more likely to be eliminated first, and to reduce the number of frozen variables. After that, the variables are ordered by a score in ascending order (line 14). In the following part, E represents the set of elected variables, and F represents the set of frozen variables. The variables are iterated through until either none are left, or the heuristic cutoff (μ) for the number of clauses in the occurrence list of either corresponding literals is reached (lines 17-37). Note that this loop is executed sequentially. If the variable is in the frozen set F, the variable is skipped and the next iteration begins. If the variable is not in the frozen set, then it is included in the elected variable set E. All clauses in which either literal of the elected variable occur are iterated through and the variables corresponding to all the other literals are included in the frozen set (lines 30-36).

Algorithm 18 Least Constrained Variable Election

```
1: function LCVE($F, OT, \mu$)
2:     $scores \leftarrow \emptyset$
3:     for $v \in vars(F)$ do                                  ▷ In parallel over $v$
4:         $p \leftarrow lit(v)$
5:         $n \leftarrow neg(p)$
6:         $s.var \leftarrow = v$
7:         if $count(OT[p]) = 0 \vee count(OT[n]) = 0$ then
8:             $s.value \leftarrow max(count(OT[p]), count(OT[n]))$
9:         else
10:            $s.value \leftarrow count(OT[p]) \cdot count(OT[n])$
11:        end if
12:        $scores.append(s)$
13:    end for
14:    SortByValueAsc($scores$)
15:    $E \leftarrow \emptyset$
16:    $F \leftarrow \emptyset$
17:    for $s \in scores$ do
18:        $p \leftarrow lit(s.var)$
19:        $n \leftarrow neg(p)$
20:        if $s.var \in F$ then
21:            continue
22:        end if
23:        if $count(OT[p]) = 0 \wedge count(OT[n]) = 0$ then
24:            continue
25:        end if
26:        if $count(OT[p]) > \mu \vee count(OT[n]) > \mu$ then
27:            break
28:        end if
29:        $E \leftarrow E \cup s.var$
30:        for $c \in OT[p] \cup OT[n]$ do
31:            for $l \in c$ do
32:                if $var(l) \neq s.var$ then
33:                    $F \leftarrow F \cup var(l)$
34:                end if
35:            end for
36:        end for
37:    end for
38:    return $E$
39: end function
```

Back to the main stage, HSE and BCE are performed partially on the elected variables, only if Stage 3 has been disabled. BVE is performed on the elected variables in parallel (line 7). The BVE method (Algorithm 19) works as follows. Firstly, resolution is performed on the selected variable, followed by the heuristic test. If the test fails, the operation is aborted (line 6). After this, the clauses used for resolution are eliminated, and the resolvents are added to the formula.

Algorithm 19 Bounded Variable Elimination

```
function BVE($x, F, OT$)
    $C \leftarrow F[OT[x]]$
    $D \leftarrow F[OT[\neg x]]$
    $CxD \leftarrow$ Resolution($x, C, D$)
    if $length(C) + length(D) < length(CxD)$ then
        return $\emptyset$
    end if
    for $c \in C \cup D$ do
        $c.deleteLater \leftarrow true$
    end for
    $F \leftarrow F \cup CxD$
end function
```

The resolution method (Algorithm 20) works by performing partial resolution on pairs between two sets of clauses on the given variable. The resolvents are accumulated and returned. Tautological resolvents are discarded.

Algorithm 20 Resolution

```
function Resolution($x, C, D$)
    $R \leftarrow \emptyset$
    for $c \in C$ do
        for $d \in D$ do
            $r \leftarrow$ PartialResolution($x, c, d$)
            if $r \neq \emptyset$ then
                $R.insert(r)$
            end if
        end for
    end for
    return $R$
end function
```

The partial resolution method (Algorithm 21) works as follows. The clauses are copied (lines 2-3) and the resolution variable is eliminated from both copied clauses (lines 4-5). After that, they are merged together into the resolvent preserving the absolute order of the literals (lines 9-28). Finally, the resolvent is checked for tautological redundancy (line 29).

Algorithm 21 Partial Resolution

```
1: function PartialResolution($x, c, d$)
2:     $c' \leftarrow copy(c)$
3:     $d' \leftarrow copy(d)$
4:     $c'.erase(x)$
5:     $d'.erase(\neg x)$
6:     $a \leftarrow c'.begin()$
7:     $b \leftarrow d'.begin()$
8:     $r \leftarrow \emptyset$
9:     while $a \neq c'.end() \wedge b \neq d'.end()$ do
10:        if $a < b$ then
11:            $r.insert(a)$
12:            $a.next()$
13:        else
14:            $r.insert(b)$
15:            if $a = b$ then
16:                $a.next()$
17:            end if
18:            $b.next()$
19:        end if
20:    end while
21:    while $x \neq c'.end()$ do
22:        $r.insert(x)$
23:        $x.next()$
24:    end while
25:    while $y \neq d'.end()$ do
26:        $r.insert(y)$
27:        $y.next()$
28:    end while
29:    if TE($r$) then
30:        return $\emptyset$
31:    end if
32:    return $r$
33: end function
```

7 Implementation Details

In this section, the technical details concerning the implementation of the algorithm are going to be presented and discussed, as well as the solving framework upon which the algorithm was being built.

The implementation was written in the C++ language. For multi-threading, the `std::thread` class from the *C++11* standard was used. The *OpenMP* library [17] was also considered, but the chosen approach proved to be more flexible and have less overhead, at least in a limited number of tests. The *MSVC* compiler and debugger tools were used for development, along with the *Microsoft Visual Studio* development environment. The source code is available in a public repository [1].

7.1 ParaFROST

The Parallel Formal Reasoning Of SaTisfiability (ParaFROST) SAT solver is developed by M. Osama et. al. [20]. In essence, it is a CDCL-based SAT solver capable of performing GPU accelerated simplifications during preprocessing and inprocessing. This embedded simplifier is the successor of SIGmA. As an alternative for the GPU-based simplifier, ParaFROST also comes with a sequential CPU-only version. The latter represents the starting point onto which the algorithm presented in this paper was built.

[1] `https://github.com/vapopescu/ParaFROST`

The original simplification algorithm is presented in Algorithm 22 and works as follows: Firstly, the CDCL decision tree of the solver is backtracked in order to simplify the formula on the root of the tree. After that, BCP is executed and both original and learnt clauses are extracted in a simplification CNF (SCNF). Then, the simplification process happens repeatedly in a number of phases (lines 6-20). During a phase, the SCNF is purged of deleted clauses if necessary. Then the OT is built. After that, the Least Constrained Variable Election (LCVE) procedure is run in order to elect independent variables for processing. Then, The Occurrence Lists (OL) corresponding to the elected variables are sorted by clause length for optimisation purposes. After that, Hybrid Subsumption Elimination (combination of SE and SSE), BVE and BCE are performed on the elected variables and the clauses in their OLs. In the last phase of simplification, only ERE is performed. At the end of each simplification phase, the LCVE cutoff factor is increased. Finally, after all simplification phases have been completed, the SCNF clauses are written back to the proper CNF, used by the solver.

Algorithm 22 Simplification algorithm (original)

```
1: function SIGMIFY($cnf$)
2:     BACKTRACK( )
3:     BCP($cnf$)
4:     $scnf \leftarrow$ EXTRACT($cnf$)
5:     $\mu \leftarrow \mu_{init}$
6:     for $i \in [1, numPhases]$ do
7:         RESIZECNF($scnf$)
8:         $ot \leftarrow$ CREATEOT($scnf$)
9:         PROPAGATE( )
10:        $pvs \leftarrow$ LCVE($scnf, ot, \mu$)
11:        SORTOT($ot, pvs$)
12:        if $i < numPhases$ then
13:            HSE($scnf, ot, pvs$)
14:            BVE($scnf, ot, pvs$)
15:            BCE($scnf, ot, pvs$)
16:        else
17:            ERE($scnf, ot, pvs$)
18:        end if
19:        $\mu \leftarrow \mu * 2$
20:    end for
21:    $cnf \leftarrow$ WRITEBACK($scnf$)
22: end function
```

7.2 WorkerPool

The main method through which parallelism is achieved is with the help of the WORKERPOOL class. The class members include:

- *workers* - An array of `std::thread` objects.
- *jobQueue* - A vector of `std::function` objects representing lambda functions of jobs to be executed by the workers.
- *mutex* - A `std::mutex` used in conjunction with the condition variables.
- *workerCV* - A `std::condition_variable` used by the workers to wait for new jobs.
- *poolCV* - A `std::condition_variable` used by the master thread to join the worker threads.
- *terminate* - A `bool` flag used when destructing the pool.
- *waiting* - An `int` representing the number of worker threads not executing a job.
- *maxBatch* - An `int` representing the maximum number of jobs that can be batched together.

Listing 1 WorkerPool::init() function

```cpp
inline void init(unsigned int threads, unsigned int maxBatch)
{
  _workers.clear();
  _jobQueue.clear();
  _terminate = false;
  _waiting = 0;
  _maxBatch = maxBatch;
  if (threads == 0) threads = 1;

  for (unsigned int i = 0; i < threads; i++) {
    _workers.push_back(std::thread([this] {
      while (true) {
        std::unique_lock<std::mutex> lock(_mutex);
        std::function<bool()> condition = [this] {
          return !_jobQueue.empty() || _terminate;
        };
        if (!condition()) {
          _waiting++;
          _poolCV.notify_one();
          _workerCV.wait(lock, condition);
          _waiting--;
        }

        if (_terminate) break;
        Job job = std::move(_jobQueue.back());
        _jobQueue.pop_back();
        lock.unlock();

        job();
      }
    }));
  }
}
```

During initialization (see Listing 1), a number of worker threads are created based on an an option in the program (by default, it is the number of logical processors of the machine) and stored in the *workers* array. Each worker waits for a job to be pushed to the *jobQueue* vector, then pulls a job and executes it, after which it loops back to waiting. The waiting mechanism is facilitated by the *mutex* and the *workerCV* condition variable which waits on the condition that the *jobQueue* is not empty or the *terminate* flag is true. At destruction (see Listing 2), the *terminate* flag is set, which breaks the loop of the worker threads. Then the worker threads are joined to the master thread and all class members are unset.

Listing 2 WorkerPool::destroy() function

```
inline void destroy() {
  if (_terminate) return;
  _terminate = true;
  _workerCV.notify_all();

  for (auto& w : _workers) {
    w.join();
  }

  _workers.clear();
  _jobQueue.clear();
}
```

The *join*() function (see Listing 3) is used to make the master thread to wait for the worker threads to finish all the jobs in the *jobQueue*. The wait is done using the *mutex* and the *poolCV* condition variable which waits on the condition that the *jobQueue* is empty and *waiting* is equal to the size of the *workers* array. Note that this function does not join the worker threads in the sense of the `c++` standard, but only in an algorithmic way. In that sense, the worker threads are only truly joined when the pool is destroyed.

Listing 3 WorkerPool::join() function

```
inline void join() const {
  std::unique_lock<std::mutex> lock(_mutex);
  _poolCV.wait(lock, [this] {
    return _jobQueue.empty() && _waiting == _workers.size();
  });
}
```

The *doWork*() function (see Listing 4) takes a lambda function as an argument and copies it to the *jobQueue*, one for each worker thread. As a result, each worker thread executes the same function independently of each other. In practice, this function is used in conjunction with mutexes, a dynamic work queue declared outside the WorkerPool class and condition variables in order to execute work on elements in a dynamic queue that has elements added inside the lambda function.

Listing 4 WorkerPool::doWork() function

```
inline void doWork(const Job& job)
{
  std::unique_lock<std::mutex> lock(_mutex);

  for (unsigned int i = 0; i < _workers.size(); i++) {
    _jobQueue.push_back(job);
  }

  _workerCV.notify_all();
}
```

The *doWorkForEach*() function (see Listing 5) takes an integer range and a lambda function as parameters. It also takes an integer representing the maximum number of jobs a batch can have. The function splits the integer range into batches of jobs as evenly as possible. If the length of the range divided by the number of workers is less or equal to the maximum allowed batch size, then the range is divided in equal batches of jobs (almost equal if the size of the range is not exactly divisible), one batch for each worker. If the range is too big, then it is divided in a number of batches of maximum size, plus another one with the remainder. The workers process these batches in parallel until none remain. The design choice behind the maximum batch size parameter is the necessity to tune the granularity of the range division in a case by case basis, where the expected running time of the lambda function has high variance amongst different values in the range (e.g. during clause elimination, the number of literals in the clause has a large impact).

Listing 5 WorkerPool::doWorkForEach() function

```
template<class IntType, class Function>
inline void doWorkForEach(const IntType& begin, const IntType& end, const IntType&
    ↪  maxBatch, const Function& job)
{
  std::unique_lock lock(_mutex);
  IntType idx = begin;
  IntType batchSize = (end - begin) / (IntType)_workers.size();
  IntType remainder = (end - begin) % (IntType)_workers.size();
  if (maxBatch > 0 && batchSize > maxBatch) {
    batchSize = maxBatch;
    remainder = 0;
  }

  while (idx < end) {
    IntType thisBatchSize = idx < remainder ? batchSize + 1 : batchSize;
    if (idx + thisBatchSize > end) thisBatchSize = end - idx;
    _jobQueue.push_back([this, idx, thisBatchSize, job] {
      for (IntType j = idx; j < idx + thisBatchSize; j++)
        job(j);
    });
    idx += thisBatchSize;
  }

  _workerCV.notify_all();
}
```

8 Benchmarks

In this section, the methodology of the benchmark procedure will be presented. The configurations used as data points will also be detailed and the results of each one will be presented and discussed.

8.1 Methodology

The procedure for benchmarking is based on the TACAS-21 Artefact of the ParaFROST solver [2]. It includes the CPU and GPU versions of the ParaFROST solver [20] and also the CaDiCaL solver [4] by A. Biere. The ParaFROST CPU solver has been updated such as to replace the simplifier algorithm to the one described in this paper. Modifications have been made to the artefact engine in order to be able to select which solver to run and the number of worker threads to utilize.

The CNF formulas on which the artefact was run are comprised by a collection of formulas presented at recent SAT competitions [3]. The formulas have been classified by the expected solving time. For this paper, only the 'easy' class of formulas has been run with the timeout of 500 seconds.

One benchmarking experiment consists of running the artefact on the same solver and configuration 5 times. The result with the median total solving time is then taken into consideration. The time

[2] `https://gears.win.tue.nl/software/parafrost/`
[3] `https://gears.win.tue.nl/software/parafrost/`

aggregates (simplification, solving, and total time) is calculated by taking the arithmetic mean of the times for each formula that was solved by all solvers and configurations. The PAR-2 score is represented by the sum of all solving times, where double the timeout value (i.e. 1000 seconds) was taken for unsolved instances. The cactus plot is obtained by by computing a scatter plot for the total solving times after sorting them in ascending order. The results of each experiment and the calculated aggregates can be found in the spreadsheet here [4].

The artefact was run on the Mathematics and Computer Science (MSC) cluster of the TU/e High Performance Computing (HPC) server platform [5]. The CaDiCaL and ParaFROST CPU solvers were run on nodes with *Intel Xeon Platinum 8260* processors (2.4GHz base-clock, 3.9GHz turbo-clock and hyper-threading enabled) and 256GB of system memory. The ParaFROST GPU solver was run on a node with an *Xeon Gold 6134* central processor (3.2GHz base-clock, 3.7GHz turbo-clock and hyper-threading enabled) with 256GB of system memory and an *nVIDIA Tesla V100* graphics processor with 16GB of dedicated memory.

8.2 Configurations

- CaDiCaL - `./cadical -t 500 -n -P5 --cover=true --elimboundmax=32`
  This is the baseline for CPU solving. It runs sequentially and is deterministic.
- ParaFROST GPU - `./parafrost --timeout=500 --workers=N -sigma -sigmalive`
  This a reference point to compare against when the simplyfying is done on the GPU. The executable is compiled separately from the CPU version.
- ParaFROST CPU - `./parafrost --timeout=500 --workers=N -sigma -sigmalive -no-bce -no-ce -no-igr -no-hbr`
  This the basic (barebones) configuration. Both pre- and inprocessing are enabled, but they run only BVE, HSE, and ERE. Each experiment is run with a different number of worker threads enabled, namely 1, 4, 16, and 48. Keep in mind that the 48 worker configurations have Hyper-Threading enabled, so two worker threads run on the same physical core.
- ParaFROST CPU (BCE) - `./parafrost --timeout=500 --workers=N -sigma -sigmalive -bce -no-ce -no-igr -no-hbr`
  Compared to the basic configuration, BCE is also enabled.
- ParaFROST CPU (IGR) - `./parafrost --timeout=500 --workers=N -sigma -sigmalive -no-bce -no-ce -igr -no-hbr`
  Compared to the basic configuration, the IGR stage is enabled without Hyper-Resolution.
- ParaFROST CPU (HBR) - `./parafrost --timeout=500 --workers=N -sigma -sigmalive -no-bce -no-ce -igr -hbr`
  Compared to the basic configuration, the IGR stage is enabled with Hyper-Resolution.
- ParaFROST CPU (CE) - `./parafrost --timeout=500 --workers=N -sigma -sigmalive -bce -ce -no-igr -no-hbr -no-rse`
  Compared to the basic configuration, the CE stage is enabled, using the standard SE technique, and no HLA or ALA.
- ParaFROST CPU (RSE) - `./parafrost --timeout=500 --workers=N -sigma -sigmalive -bce -ce -no-igr -no-hbr -rse`
  Compared to the basic configuration, the CE stage is enabled, using the RSE technique, and no HLA or ALA.

8.3 Results

In Figure 2, we take a look at the cactus plot of some experiments. On the x-axis we have the index of the formulas sorted in ascending order individually for each dataset. It has been cut to show the formulas with higher running times in order to improve readability. On the y-axis we have the total solving time.

[4] https://docs.google.com/spreadsheets/d/1C4Mt4Vlmi8JvxgaFjSIUP9Q9ANM7IGLf/view
[5] https://hpcwiki.tue.nl/wiki/

For the CPU configurations, only the experiments running with 48 workers, and the basic configuration with 1 worker are shown. The rest are omitted in order to increase visibility. First in line is the CaDiCaL solver, our baseline. Next is the HBR configuration performing only slightly better, but managing to solve a couple less formulas. Then we have the rest of the CPU configurations forming a tight cluster, with the IGR and BCE configurations running slightly worse overall, and the basic configuration slightly better. Finally, the GPU version was the best performing of them all, managing to solve 108 out of 109 formulas, and doing so the fastest overall.

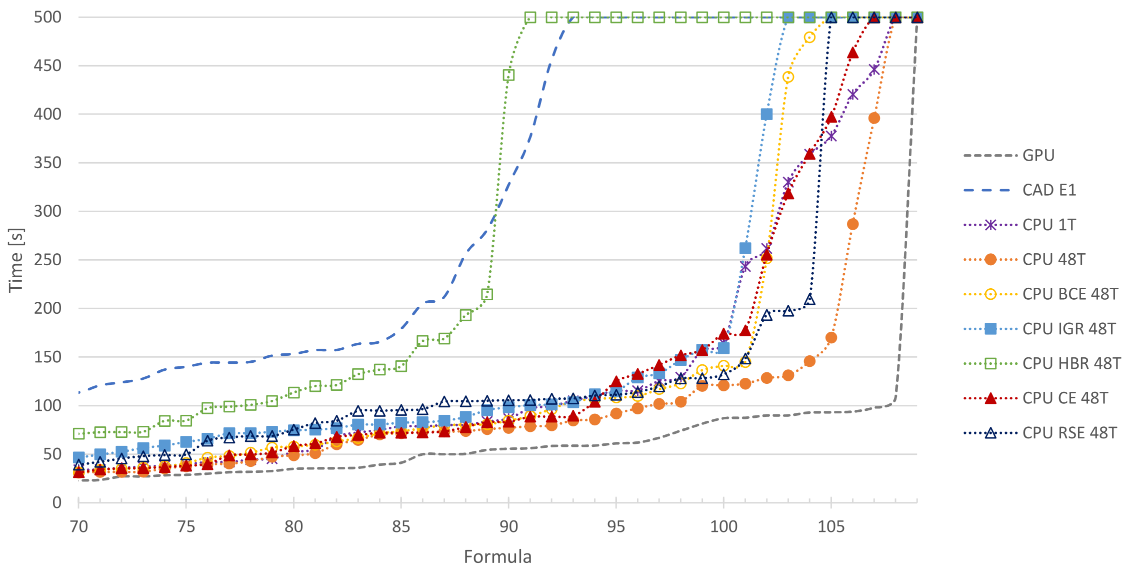

Figure 2: Cactus plot for total time.

A similar story as before can be seen on the PAR-2 score graph in Figure 3. Here, on the x-axis we have the configured worker count, and on the y-axis we have the PAR-2 score as defined above. We can also see that the high number of timeouts is also having an effect here for the HBR configuration. What is strange is that the experiment running 4 worker threads managed to solve way less formulas. This could be due to the fact that a single worker finding new edges in the graph can restart immediately, while there is a small overhead in waiting for all the threads to abort HBR method which accumulates over time, having cascading effects. However we can see a decrease in the score once we increase the number of workers, which is more in line with the expectations. Also in line with the expectations is the IGR configuration where the score decreasing when increasing the number of workers, up to 16. The higher score for the 48 worker experiment can be explained by the overhead induced by Hyper-Threading. The seemingly constant score of the basic CPU and BCE configuration up to 16 workers can be explained by the number of formulas solved (which is roughly the same) and the solving time having a higher influence on the score than the simplification time.

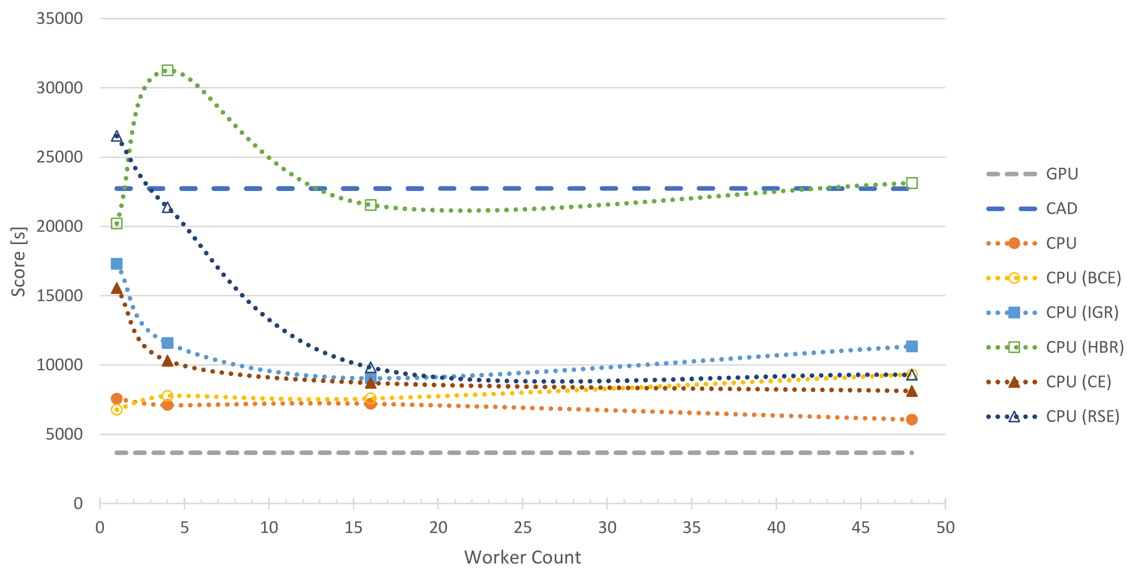

Figure 3: Benchmark results for PAR-2 score.

Moving on to the next graphs, only the largest subset of formulas solved by all configurations is considered. On the x-axis we have the number of workers for the configurations (CaDiCaL and ParaFROST GPU are shown as an horizontal line), and on the y- axis we have the aggregate times as arithmetic means for each configuration. For the simplification time (Figure 4), we see a general decrease in processing time as the number of workers increases. In terms of speedup compared to the 1 worker configuration, the basic configuration was 1.06/1.67/1.74 times faster on average for the 4, 16, and 48 worker configurations respectively. The BCE configuration ran 1.01/1.60/1.58 times faster, while the IGR configuration ran 1.09/1.43/1.46 times faster. The HBR configuration ran 1.35/1.12/1.15 times faster, with the 4 worker configuration being an anomaly once again. Also, the HBR implementations seems to not scale very well with the number of workers, suggesting that a different approach is needed for this technique. The CE and RSE ended up performing better than expected, scaling the best with the number of workers, with 1.81/3.47/4.54 and 1.09/2.12/2.65 speedup on average respectively. This is contrary to the expectations given by the preliminary testing. This is due to the processor of the testing machine having many more cores compared to the development machine (48 threads vs. 16 threads).

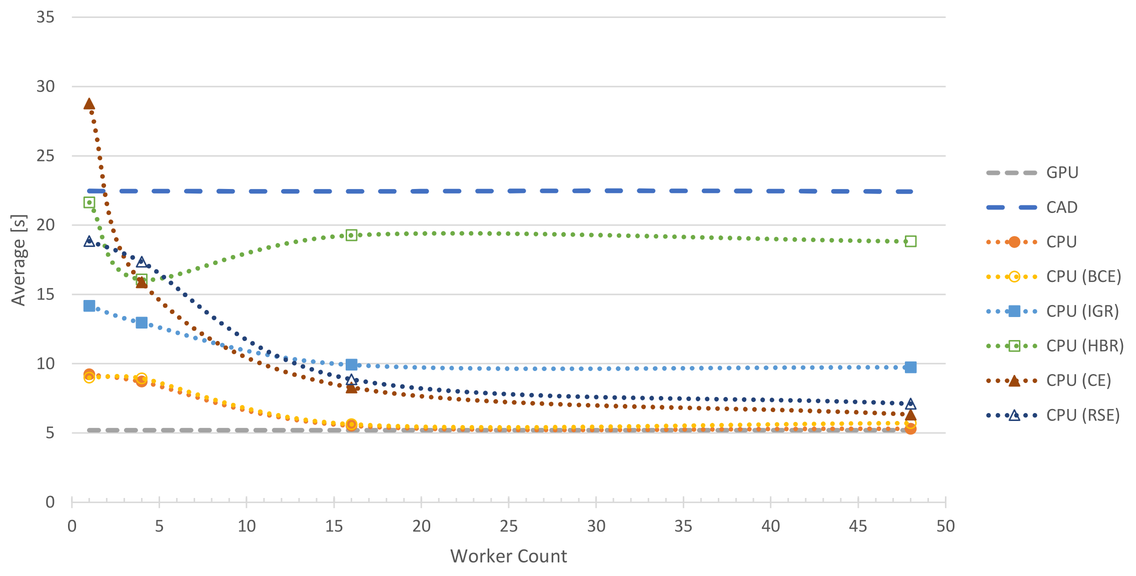

Figure 4: Benchmark results for simplification time.

For the next graph, the expectation was for the solving time to be independent and consistent in relation to the number of workers used, since the solving algorithm is running sequentially on one thread. What we got in Figure 5 however, are seemingly inconsistent solving times. As such, no clear conclusions can be drawn regarding the effect each simplification technique has on the solving time. Only that the HBR configuration seems to be performing better in this regard.

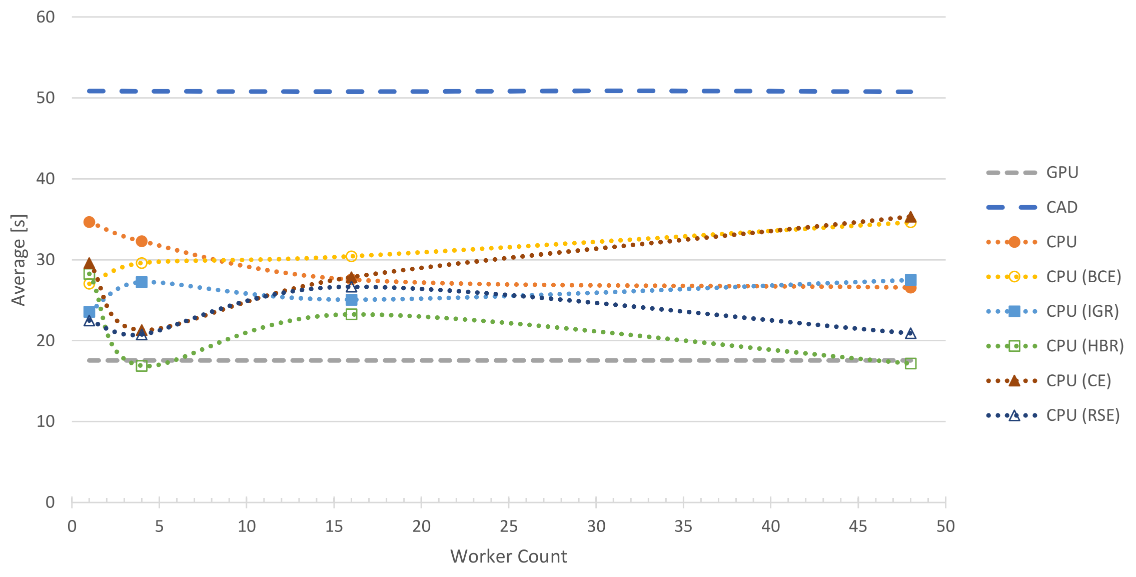

Figure 5: Benchmark results for solving time.

Regarding the total solving time in Figure 6, we can see the inconsistent solving times also having an effect here. However we can still make a few remarks. Firstly, the HBR configuration running with 4 threads is an anomaly once again. Then the trend of the running time seems to be generally downwards for the rest of the configurations, up until 16 workers. At 48 workers, the behaviour is much too inconsistent.

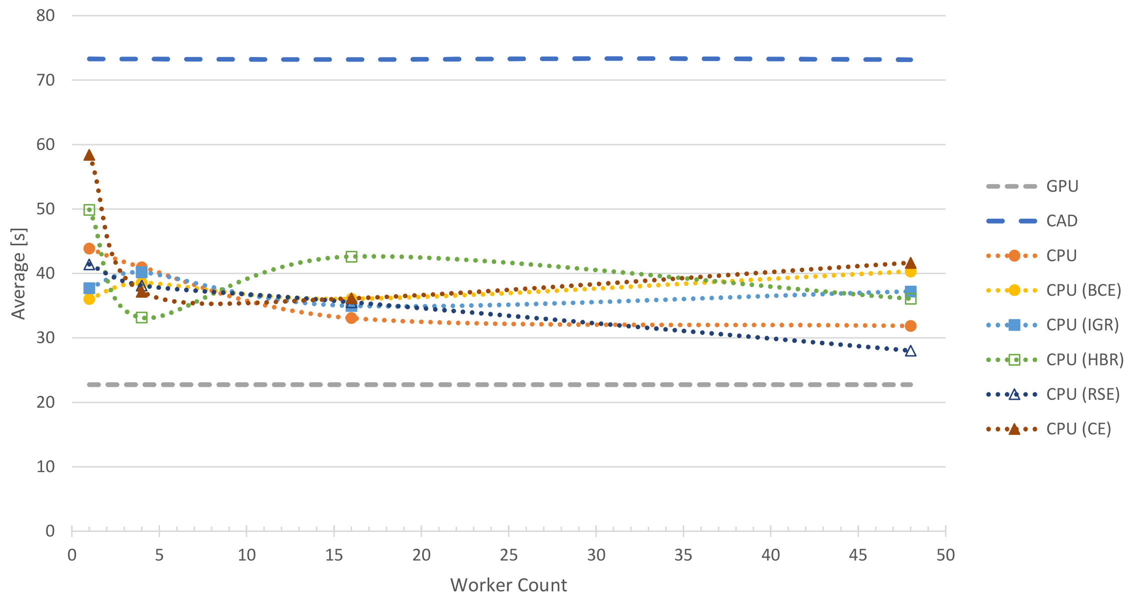

Figure 6: Benchmark results for total time (simplification + solving).

8.4 Threats to Validity

The benchmark results in this section were validated in the sense that the output of the solver either matched the baselines set by the ParaFROST GPU [20] and CaDiCaL [4] solvers in terms of the satisfiability (or unsatisfiablility) of the formula, or timed out. However, the outputs of the solver with the simplifier presented in this paper were not proof checked in the case of unsatisfiability, or checked for correct assignments in the case of satisfiability.

Some experiments have also failed to run. More specifically, the for IGR configuration with 48 workers, and the HBR configuration with 16 and 48 workers. The exact issue causing this was investigated, but not found. The results presented above are taken from experiments that did manage to run and complete.

9 Conclusion

In this paper, a survey of simplification techniques was presented, along with ways in which they can be parallelized to run on multi-threaded CPUs. Using these techniques in a configurable manner, an algorithm was proposed in order to perform SAT simplification in a multi-threaded way. An implementation of the said algorithm was then benchmarked and the results were shown.

Regarding the benchmarks, we saw a benefit by running the different configurations of the simplification algorithm with multiple workers. The basic, BCE and IGR showed consistent improvements with increasing number of workers. However, the HBR configuration either showed highly inconsistent behaviour, or suffered from poor scaling. This suggests that the implementation presented here is not optimal, at least for multi-threading use. The CE and RSE configurations scaled the best, although they were the worst performing with the single worker configuration. However, no configuration was able to achieve comparable results to the GPU implementation by M. Osama and A. Wijs [20].

Based on the presented results, we make some recommendations. Firstly, we suggest to keep the number of workers tied to the number of logical processors in the machine, as those configurations showed the most benefit. We also advise disabling BCE, as it showed worse solving times overall when compared to the basic configuration. We also recommend enabling CE only on machines with 16 or more processor threads, as the running time is too high otherwise. Since it showed worse results overall than CE, RSE should be disabled. IGR and HBR should be disabled, as they showed worse total running times and fewer solved formulas respectively. Finally, if a GPU accelerator is available to you, use the corresponding version of the simplifier instead [20].

9.1 Future Work

There are ways in which the work presented in this paper may be built upon. Firstly, the benchmark results may be confirmed using a proof checker tool. Regarding the implementation, certain areas can be improved. Specifically, the operation of the worker threads is left to be handled by the Operating System, as this solution was meant to be platform independent. If it is desired however, platform-specific methods may be employed to reduce the overhead of thread switching (and cache flushing as an effect). Lastly, other simplification techniques not discussed here could be added to the algorithm.

10 References

[1] Fahiem Bacchus. “Enhancing Davis Putnam with Extended Binary Clause Reasoning”. In: *Eighteenth National Conference on Artificial Intelligence.* Edmonton, Alberta, Canada: American Association for Artificial Intelligence, 2002, pp. 613–619. ISBN: 0262511290.

[2] Fahiem Bacchus and Jonathan Winter. “Effective Preprocessing with Hyper-Resolution and Equality Reduction”. In: *Theory and Applications of Satisfiability Testing.* Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 341–355. ISBN: 978-3-540-24605-3.

[3] Armin Biere et al. “Bounded Model Checking.” In: *Handbook of satisfiability* 185.99 (2009), pp. 457–481.

[4] 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.

[5] Armin Biere et al. “Symbolic Model Checking without BDDs”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-49059-3.

[6] Vincent Bloemen and Jan Cornelis van de Pol. “Multi-core SCC-Based LTL Model Checking”. Undefined. In: *Hardware and Software: Verification and Testing; Proceedings of the 12th International Haifa Verification Conference, HVC 2016.* Ed. by Roderick Bloem and Eli Arbel. Lecture Notes in Computer Science. 10.1007/978-3-319-49052-6_2 ; null ; Conference date: 14-11-2016 Through 17-11-2016. Netherlands: Springer, Nov. 2016, pp. 18–33. ISBN: 978-3-319-49051-9. DOI: `10.1007/978-3-319-49052-6\_2`.

[7] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing.* STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: `10.1145/800157.805047`. URL: `https://doi.org/10.1145/800157.805047`.

[8] Martin Davis, George Logemann, and Donald Loveland. “A Machine Program for Theorem-Proving”. In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: `10.1145/368273.368557`. URL: `https://doi.org/10.1145/368273.368557`.

[9] Niklas Eén and Armin Biere. “Effective Preprocessing in SAT Through Variable and Clause Elimination”. In: *Theory and Applications of Satisfiability Testing.* Ed. by Fahiem Bacchus and Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 61–75. ISBN: 978-3-540-31679-4.

[10] Kilian Gebhardt and Norbert Manthey. “Parallel Variable Elimination on CNF Formulas”. In: *KI 2013: Advances in Artificial Intelligence.* Ed. by Ingo J. Timm and Matthias Thimm. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 61–73. ISBN: 978-3-642-40942-4.

[11] Roman Gershman and Ofer Strichman. “Cost-Effective Hyper-Resolution for Preprocessing CNF Formulas”. In: *Theory and Applications of Satisfiability Testing.* Ed. by Fahiem Bacchus and Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 423–429. ISBN: 978-3-540-31679-4.

[12] Marijn Heule, Matti Järvisalo, and Armin Biere. “Clause Elimination Procedures for CNF Formulas”. In: *Logic for Programming, Artificial Intelligence, and Reasoning.* Ed. by Christian G. Fermüller and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 357–371. ISBN: 978-3-642-16242-8.

[13] Matti Järvisalo. “Equivalence checking hardware multiplier designs”. In: *Papers on Computational Logic* 1967 (1970), pp. 466–483.

[14] Matti Järvisalo, Armin Biere, and Marijn Heule. “Blocked Clause Elimination”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Javier Esparza and Rupak Majumdar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 129–144. ISBN: 978-3-642-12002-2.

[15] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. “Inprocessing Rules”. In: *Automated Reasoning.* Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 355–370. ISBN: 978-3-642-31365-3.

[16] Leonid A. Levin. “Universal Sequential Search Problems”. In: *Problems of Information Transmission* 9.3 (1973).

[17] Tim Mattson. “An introduction to OpenMP”. In: Feb. 2001, pp. 3–3. ISBN: 0-7695-1010-8. DOI: `10.1109/CCGRID.2001.923161`.

[18] Muhammad Osama and Anton Wijs. “Parallel SAT Simplification on GPU Architectures”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 21–40. ISBN: 978-3-030-17462-0.

[19] Muhammad Osama and Anton Wijs. “SIGmA: GPU Accelerated Simplification of SAT Formulas”. In: *Integrated Formal Methods.* Ed. by Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa. Cham: Springer International Publishing, 2019, pp. 514–522. ISBN: 978-3-030-34968-4.

[20] Muhammad Osama, Anton Wijs, and Armin Biere. “SAT Solving with GPU Accelerated Inprocessing”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Cham: Springer International Publishing, 2021, pp. 133–151. ISBN: 978-3-030-72016-2.

[21] Muhammad Osama et al. “An Efficient SAT-Based Test Generation Algorithm with GPU Accelerator”. In: *Journal of Electronic Testing* (Oct. 2018), p. 17. DOI: 10.1007/s10836-018-5747-4.

[22] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. “NiVER: Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances”. In: *Theory and Applications of Satisfiability Testing*. Ed. by Holger H. Hoos and David G. Mitchell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 276–291. ISBN: 978-3-540-31580-3.

[23] Anton Wijs and Dragan Bošnački. “GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 233–247. ISBN: 978-3-642-54862-8.