Parallelization of an All-SAT solver

Saúl Adserias Valero

Abstract– Advancements in SAT solving algorithms with heuristics and more sophisticated inferring techniques have allowed a wide array of real world applications for SAT solvers to prosper. In this work, an All-SAT solver based on the DPLL algorithm has been developed, optimized, and ultimately parallelized, with the objective of learning about the intricacies of SAT solvers from a performance engineering perspective. Ultimately, as a result of this work, a correct and complete parallel All-SAT solver, based on task-parallelism and a divide-and-conquer paradigm, has been successfully implemented and tested with set of benchmark satisfiability problems.

Keywords– All-SAT, SAT, DPLL, solver, parallelization, tasks, CNF, unit propagation, backtracking, propositional formula.

1 Introduction

The All-SAT problem consists in finding all the satisfying assignments of a propositional formula. It is a variation of the more famous boolean satisfiability problem SAT, where only one satisfying assignment needs to be found in order to form a solution.

More formally, given a propositional logic formula $\mathcal{F}$ defined with a finite set of variables V and a finite set of logical operators that connects them, the All-SAT problem can be stated as the search of a subset $\Lambda_{\mathrm{T}} \subseteq \Lambda$, where Λ is the set of all assignment functions λ_i that have the set of variables V as the domain and the Boolean set $\mathbb{B} = \{\mathrm{T},\ \mathrm{F}\}$ (where T and F are the truth values for true and false, respectively) as the co-domain:

$$\Lambda = \left\{ \lambda_i : V \to \mathbb{B} \;\middle|\; 1 \leq i \leq 2^{|V|} \right\},$$

such that

$$\Lambda_{\mathrm{T}} = \{\lambda_i \in \Lambda \mid \mathcal{F}_{\lambda_i} = \mathrm{T}\},$$

where $\mathcal{F}_{\lambda_i}$ is the truth value for the evaluation of the formula $\mathcal{F}$ with the assignment of the variables $v_k \in V$ according to $\lambda_i(v_k)$.

It is well known that SAT belongs to the NP-complete complexity class of problems [1]. Meanwhile, since All-SAT is not a decision problem, it can't be categorized the same way as its counterpart. It can be said though, that it is most proximal to a counting problem like #SAT, a #P-complete problem that consists in giving the total number of possible satisfying assignments, for a given formula (but not the assignments themselves). It can be argued then that All-SAT is harder than SAT but pretty similar to #SAT.

The relevance of SAT and it's variants in computer science and mathematics has fueled the development of a class of software programs specialized in solving the problem [2]. They are commonly referred to as SAT solvers.

Remarkably, even though only algorithms with worst case exponential complexity exist, actual solver implementations use multitude of heuristics and other techniques to overcome this complexity limitation, making them useful for many real-world problems from diverse areas of knowledge such as artificial intelligence and electronic design automation. Some of this strategies will be later discussed.

With regards to this work, the focus has been set on the development of a complete All-SAT solver based on the systematic search of the solution space. The aim was to learn while implementing some of the techniques that form it, to later optimize them and ultimately parallelize the whole solver.

Following is section 2, where some preliminary knowledge for the context of the work is given. Next is section 3, that will address all about the development of the solver and the proposed versions. In section 4 we will present the obtained results of the proposals, and finally in section 5 some conclusions will be laid out to wrap the work.

2 Preliminaries

In this section some notation and the definition of the DPLL algorithm, along with the current state of the art for SAT solvers, is presented.

2.1 Conjunctive normal form

For standardisation and performance purposes, most solvers expect the input problem to be encoded in conjunctive normal form, or CNF for short. A CNF form of the problem will be implicitly expected in the next sections.

A propositional formula $\mathcal{F}$ of n variables, defined as a 2-tuple of the sets $(V,\ C)$, is in conjunctive normal form if it is formed as the conjunction (denoted by $\wedge$) of m clauses

• E-mail de contacte: saul.adserias@autonoma.cat
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Juan Carlos Moure (DACSO)
• Curs 2022/23

$c_j \in C$:

$$\mathcal{F} = c_1 \wedge c_2 \wedge \ldots \wedge c_m,$$

where each clause c_j is a disjunction (denoted by $\vee$) of n_{c_j} literals l_x:

$$c_j = l_1 \vee l_2 \vee \ldots \vee l_{n_{c_j}},$$

letting a literal l_x be the truth value of a variable $v_k \in V$ ($l_x = v_k$) or it's negation ($l_x = \neg v_k$).

Crucially, a formula $\mathcal{F}$ in CNF form is satisfied if each of the clauses is independently satisfied (i.e. has a literal that evaluates to the T truth value). On the other hand, as a byproduct, if a clause can't be satisfied, so can't be $\mathcal{F}$.

As an example, the following formula $\mathcal{G} = (V,\ C)$:

$$\begin{aligned}\mathcal{G} = &(A \vee B)\\ \wedge\ &(B \vee C)\\ \wedge\ &(\neg A \vee \neg X \vee Y)\\ \wedge\ &(\neg A \vee X \vee Z)\\ \wedge\ &(\neg A \vee \neg Y \vee Z)\\ \wedge\ &(\neg A \vee X \vee \neg Z)\\ \wedge\ &(\neg A \vee \neg Y \vee \neg Z)\end{aligned} \tag{1}$$

is in CNF, and is defined by the variable set

$$V = \{A,\ B,\ C,\ X,\ Y,\ Z\}$$

of size $|V| = n = 6$ and the clause set

$$\begin{aligned}C = \{&\{A,\ B\},\\ &\{B,\ C\},\\ &\{\neg A,\ \neg X,\ Y\},\\ &\{\neg A,\ X,\ Z\},\\ &\{\neg A,\ \neg Y,\ Z\},\\ &\{\neg A,\ X,\ \neg Z),\\ &\{\neg A,\ \neg Y,\ \neg Z\}\}\end{aligned}$$

of size $|C| = m = 7$, where $|c_{1,2}| = 2$ and $|c_{3,4,5,6,7}| = 3$.

2.2 Additional notation

Taking some freedom in the definition and notation previously presented, assignment functions will sometimes be only partially defined ($\lambda : V \rightharpoonup \mathbb{B}$). Also, this kind of functions will sometimes be expressed as a set of ordered pairs $\lambda = \{(v,\ b) \mid v \in V,\ b \in \mathbb{B}\}$, aside from the normal function notation. The two notations will be used interchangeably, depending on the context. Additionally, the expression $v := b$ for any $v \in V$ and $b \in \mathbb{B}$ will be shorthand for $\lambda(v) = b$.

Furthermore, we also define $\phi : L \to V$ to be the function from the set of literals $L = \{v,\ \neg v \mid v \in V\}$ to the set of variables V that maps each literal to it's corresponding variable, and $\delta : L \to \mathbb{B}$ the function that maps a literal to the truth value T or F if its polarity is positive or negative (i.e. negated variable).

Finally, as each clause $c \in C$ of a formula $\mathcal{F}$ can also be seen as a formula itself, the statement c_λ will also be the truth value of c when applying the assignment λ to its occurring variables. Furthermore, a clause $c \in C$ will also be treated as subset of the set of literals ($c \subset L$).

2.3 Basic algorithm: DPLL

The first notion of a SAT solver was developed by Davis, Putname, Longemann and Loveland in the 1960s with the DPLL algorithm [3, 4]. This is an algorithm based on backtracking that performs a search of the solution space in a depth first fashion.

As a brief overview, it can be divided in three main steps:

- the **decision** step, which consists on picking and assigning an unassigned variable (the "decision variable"), which determines the "decision level";
- the **unit propagation** step, where clauses that have only one occurring literal with an unassigned variable (called "unit clauses"), are searched to forcefully assign the corresponding variable (the "propagated variable") so that the literal, and therefore the clause, evaluates to T (since it is the only way to satisfy the clause, inferring the assignment);
- and finally the **backtracking** step, where the state of the program is restored to the last valid decision level. For a SAT solver this step is only done when one or more variables have conflicting assignments in the unit propagation step. A conflict arises when it is inferred that the same variable has to be assigned to T and F at the same time, (or equivalently, when a clause that is not satisfied has no remaining literals with an unassigned variable, becoming thus unsatisfiable). Since we want to use DPLL for the All-SAT problem the backtracking will also need to be done after each solution is found.

The pseudocode in Algorithm 1 describes a recursive DPLL implementation using the UNIT-PROPAGATION function defined in Algorithm 2 and the auxiliary function CONFLICT, defined in Algorithm 3, to check for conflicts after each propagation.

For the decision step, the function DECIDE-VAR implements the picking of an unassigned variable $v \in V$ according to λ, and the assignment is done in each of the recursive calls building a new λ function, for each of the recursive calls, with the two truth values.

Finally, the backtracking step is, in this case, implicit, since the only changing state is the assignment function λ and a new one is formed for each decision level, based on a copy of the previous one, that only lives in the scope of its level ℓ.

Figure 1 represents a full example of all the steps that the DPLL algorithm makes, in conjunction with the effects on the given formula, to solve the problem.

The two initial decisions $A :=$ T and $X :=$ F (d_1 and d_2) determine by unit propagation that $Z :=$ T and $Z :=$ F (p_1 and p_2) at the same time, leading to a conflict that needs to backtrack, chronologically, to the previous valid decision level (b_1).

The new decision path (d_3) leads to another conflict, meaning that the solver needs to backtrack to the first decision level (b_2), since the second has no more valid assignments to explore.

Finally, when the decision $A := F$ is made (d_4), clauses 3 through 7 are now satisfied. At the same time, by unit propagation on the first clause, it is inferred that $B := T$

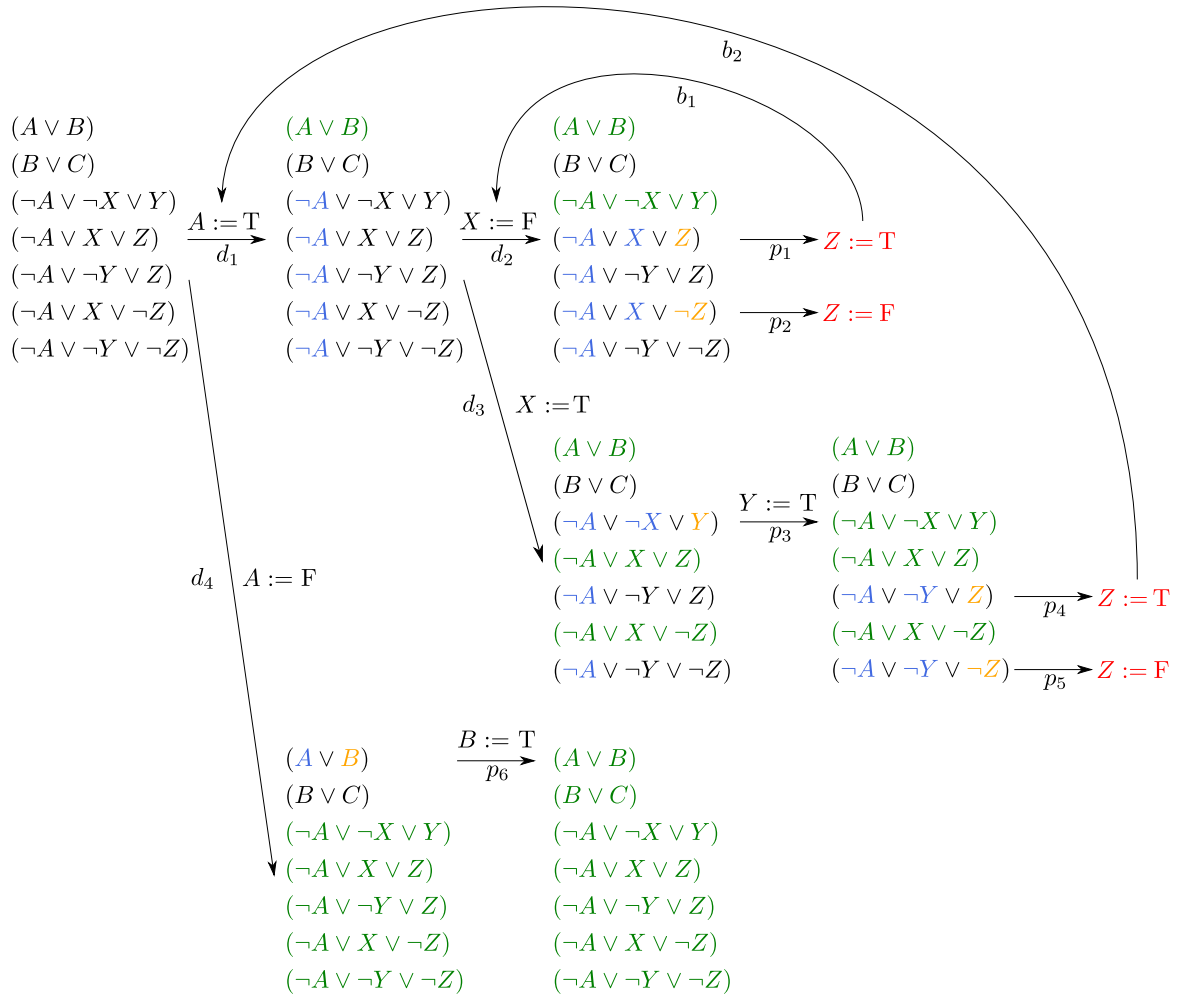

Fig. 1: Example of the full solving procedure of the formula $\mathcal{G}$ (see equation 1) following the DPLL algorithm and the decision, unit propagation and backtracking steps (denoted by the labeled arrows d_x, p_y, and b_z, respectively), where blue represents literals that have an assigned variable but evaluate to F, orange identify the remaining literals in a unit clause, and red determines propagated variable assignations (from unit propagation) that cause a conflict.

Algorithm 1 DPLL recursive procedure with $\mathcal{F}$ being the propositional formula (defined as a tuple with the set of variables V and the set of clauses C), λ the assignment function (initially $\emptyset$), and ℓ a marker of the decision level (initially 0).

```
1:  function DPLL(F = (V, C), λ, ℓ)
2:      λ_p ← UNIT-PROPAGATION(C, λ)
3:      if CONFLICT(λ_p) = T then
4:          return ∅
5:      λ ← λ ∪ λ_p
6:      if F_λ = T then
7:          return λ
8:      v ← DECIDE-VAR(V, λ)
9:      Λ_T^{λ(v)=T} ← DPLL(F, λ ∪ (v, T), ℓ + 1)
10:     Λ_T^{λ(v)=F} ← DPLL(F, λ ∪ (v, F), ℓ + 1)
11:     return Λ_T^{λ(v)=T} ∪ Λ_T^{λ(v)=F}
```

Algorithm 2 Function to propagate unit clauses in C with respect to the current assignment function λ, returning another assignment function λ_p with the propagated variables and its corresponding truth values. The $\exists!$ symbol meaning unique existence.

```
1:  function UNIT-PROPAGATION(C, λ)
2:      λ_p ← ∅
3:      for c ∈ C do
4:          if c_λ = T then
5:              CONTINUE
6:          if ∃!l ∈ c s. t. (φ(l), T / F) ∉ λ then
7:              λ_p ← λ_p ∪ (φ(l), δ(l))
8:      return λ_p
```

Algorithm 3 Function that determines if there exist conflicting assignments in λ, returning F and T respectively.

```
1:  function CONFLICT(λ)
2:      if ∃(v, T) ∈ λ ∧ ∃(v, F) ∈ λ then
3:          return T
4:      return F
```

(p_6), satisfying both remaining clauses 1 and 2. This means that, since with the partial assignment of the variables $A := F$ and $B := T$ the problem is satisfied, the variables C, X, Y and Z remain free, making a total of $2^4 = 16$ satisfying assignments that form the solution. Note that the algorithm ends here since it can't backtrack to any valid decision level anymore.

2.4 State of the art: CDCL & local search

Most modern solvers are originally based on the DPLL algorithm but with the main addition of conflict driven clause learning (abreviated CDCL) [5, 2]. This is an optimizing technique which was first introduced by the GRASP solver [6] in the late 1990s and was quickly followed and refined upon by other solvers, such as zChaff [7], MiniSAT [8] or Glucose [9] and practically all the participant solvers in the annual SAT competitions [10].

The main idea behind the technique is the derivation of a clause from the assignments that led to the conflict, to later learn it (or more precisely, learn its negation). This is useful because, essentially, a new constraint with the failed combination of assignments is appended to the problem, making sure that the same conflict is not reached again.

Another benefit of this clause derivation is that, instead of backtracking to the last valid decision level (like a standard a DPLL solver), the conflict can be analyzed to backtrack non-chronologically to the last decision level where the clause becomes satisfied, pruning part of the search space.

On the other hand, one major issue with the CDCL approach is that, while more clauses makes the solver find the solution faster (in general), the number of learned clauses can grow enough to become a performance issue. To solve this problem most solvers include deletion policies that remove learned clauses based on their usefulness (mostly using heuristics).

Aside from complete algorithms based on systematic global search, it is worth mentioning that there exists a class of non complete solvers based on local search. This type of solvers start with a given truth assignment of variables (mostly based on heuristics), and try to improve on it until a solution is found. One example of local search solvers are stochastic ones [11, 2].

The main advantage of such solvers is the speed at which a solution can be found. On the other hand, if the solver can't find one solution, there's no guarantee that it doesn't exist, making them non-suitable for applications where correctness is needed.

3 PROPOSALS

This section presents the development methodology and a description of the main ideas behind the implementation of each incremental version of the solver.

3.1 Development methodology

The approach for the development of the solver applied in this work has most resembled an iterative and incremental methodology. The common pattern was the planning of an iteration of the solver that fulfilled a series of specific requirements and objectives, followed by the implementation and subsequent testing of it. For the next iterations, the same cycle applied in an incremental fashion, improving the solver by adding new or modifying existing functionalities of it.

Three main iterations, and therefore three main versions of the solver, has been done: baseline version, optimized version and parallel version.

First, an initial minimal version of the solver was envisioned. The plan was to make a simple but yet functional version of the solver that could read CNF formulas, encode them in a useful internal representation, and also solve them with a first implementation of a DPLL based algorithm. In this version simplicity was of more importance than performance.

For the second iteration, a performance aware approach was taken, mainly improving the efficiency of the DPLL algorithm. The aim was to make the solver more efficient such that the future parallelization would be worth and feasible.

Finally, the third major version of the solver focused on the parallelization of the solver. The main idea was to devise a strategy that would distribute the work between the processors as evenly as possible.

3.2 Baseline version

3.2.1 Problem input

The baseline version implements a basic DPLL sequential solver. It first reads a CNF formula in the DIMACS file format (see appendix A.1 for more details) and loads the information into two data structures: `clss` and `clss_idxs`. Both are 1-dimensional vectors that hold, respectively, all the literals of the formula encoded as integers, where a negative integer represents a negated literal, and the indexes that bound each clause. By consequence, each variable is represented by the absolute value of the literal.

With this two vectors it is possible to access each of the literals of a given clause c_j by getting the start and end indices (assuming 0 indexing) `s = clss_idx[`$j-1$`]` and e `= clss_idx[`j`]` respectively, to access `clss[`s`]` through `clss[`$e-1$`]`. In figure 2 an example diagram of both data structures is shown.

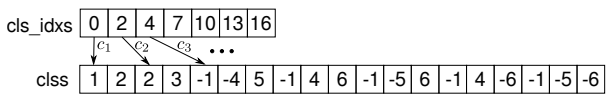

Fig. 2: Example diagram of the vector pair `clss` and `clss_idxs` for the formula $\mathcal{G}$ defined in equation 1 and encoded according to figure 5. The annotated arrows mark the first literal of each clause c_j.

3.2.2 Decision and backtracking

To model the assignment function, a specific data structure named `vars` was created. This data structure is also a vector that has the truth value assigned to the variable v_k in `vars[`$k-1$`]`. Since not all variables will be always assigned, a third truth value X was added to represent an unassigned state. This is also the starting truth value of each variable.

This data structure provides the executing state of the program. This means that, in order to backtrack, the solver needs to restore the contents of this vector to its previous state in the last valid decision level. To do so, a simple approach was to add d levels of depth to the vector, assigning each depth level to a decision level. This way, a copy with its state in each decision level could be saved independently, making the backtracking step a simple process of pointing to the correct depth level (see figure 3).

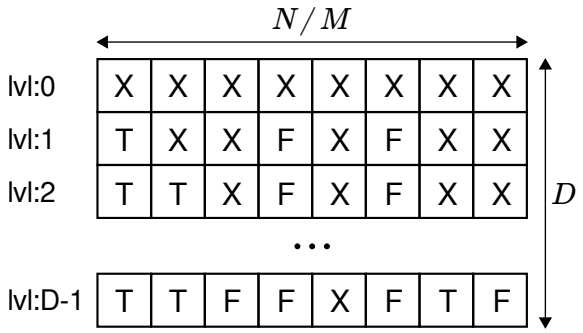

Fig. 3: Example diagram of the structure of the `vars` vector expanded with D levels of depth, each corresponding to a different decision level.

Picking a decision variable requires establishing an ordering on the `vars` structure. We use an static heuristic to prioritize the most frequent variables first. This heuristic runs after the reading of the CNF formula and changes the internal representation of each literal so that the ones that represent the most frequent variables have the lowest absolute numbers. With this implemented, an auxiliary function named `pick_var` was created in order to return a natural number representing the most frequent unassigned variable (based on the current assignation of `vars`).

3.2.3 Unit propagation

Function `unit_prop` iterates over each clause and counts the number of literals that have an assigned variable. If one of the literals satisfies the clause, the clause is discarded. There are two remaining useful cases: all the literals of the clause, but one, are assigned (a unit clause), and all the literals are assigned (a conflict).

When we identify a unit clause, an assignment for the variable of the truth value that satisfies the clause is made (i.e. $\lambda(\phi(l)) = \delta(l)$ for $l \in c$ where c is the unit clause and l its remaining literal). Then the whole unit propagation process is marked to be repeated again after passing over the remaining clauses. This is the case because, upon propagating a variable, some other unit clauses might appear, making a cascade effect of propagations.

A conflict can only happen if a unit propagation also happened before: when a unit clause is found, it is possible that another unit clause (not yet found), with an unassigned literal of the same variable, but with opposite polarity, exists. I.e., given two unit clauses c_j and c_r with the remaining unassigned literals $l_{c_j} \in c_j$ and $l_{c_r} \in c_r$, it applies that $\phi(l_{c_j}) = \phi(l_{c_r})$ and $\delta(l_{c_j}) \neq \delta(l_{c_r})$. This means that, when the first is propagated, the subsequent one will no longer be identified as a unit clause, and neither is or can be satisfied, as all its literals have an assigned variable but none evaluate to the truth value T. This is an alternative method (compare this method with the one in algorithm 3) to detect a conflict after the variable has been already propagated to either of the conflicting truth values.

A complete pseudocode implementing the described unit propagation can be seen in algorithm 4. A function named `solve` (with pretty similar structure to Algorithm 1) implements all the steps of the DPLL algorithm.

Algorithm 4 Performs unit propagation by searching for clauses in clss that only have one literal unassigned (unit clauses). Returns F if contradiction was found, T if all clauses are satisfied and X otherwise. It also uses the parameter ℓ to address the correct depth level of `vars`. The symbol $\odot$ represents an XNOR and `clss[`$s : e$`]` the range of items `clss[`j`]` $\forall j \in [s,\ e)$.

```
1:  function UNIT-PROP(ℓ)
2:      p ← T
3:      u ← T
4:      while p = T do
5:          p ← F
6:          u ← T
7:          for j ∈ [0, |C|) do
8:              u ← X
9:              n_{v≠X} ← 0
10:             l_X ← 0
11:             s ← clss_idx[j]
12:             e ← clss_idx[j + 1]
13:             for l_i ∈ clss[s : e] do
14:                 v ← vars[ℓ, abs(l_i) − 1]
15:                 if v ≠ X then
16:                     n_{v≠X} ← n_{v≠X} + 1
17:                     t ← (v = T) ⊙ (l > 0)
18:                     if t then
19:                         BREAK
20:                 else
21:                     l_X ← l_i
22:             if t then
23:                 CONTINUE
24:             u ← X
25:             if n_{v≠X} = SIZE(clss[c]) then
26:                 return F
27:             if n_{v≠X} = SIZE(clss[c]) − 1 then
28:                 if l_X > 0 then
29:                     vars[ℓ, abs(l_X) − 1] ← T
30:                 else
31:                     vars[ℓ, abs(l_X) − 1] ← F
32:                 p ← T
33:     return u
```

3.3 Optimized version

3.3.1 Unit propagation

Starting from the baseline version, a few improvements have been made with respect to the efficiency of the unit propagation process. The baseline propagation step always checks all the clauses. This is not entirely necessary, as the new unit clauses (1) must be still not satisfied, and (2) must have an occurring literal of the last decision variable. This means that, potentially, only a very small subset of clauses

has to be actually checked.

The baseline version does not memorize which clauses have been already satisfied, making the checking process very redundant. To affront this, a new data structure called `clss_sat`, conceptually identical to `vars` (see figure 3), is used to save the satisfying state of each clause, for each decision level. With this vector, analogously to a variable in `vars`, the state of a clause c_j in a given decision level ℓ would be saved in `clss_sat[`ℓ`,` $j-1$`]`, making it now possible to easily discard a clause if it was already satisfied.

A second improvement is to use a pair of vectors `var_clss` and `var_clss_idxs` that, for each variable, points to all the clauses which have an occurring literal, including its polarity. Analogously to the pair `clss` and `clss_idxs` previously described, `var_clss_idx[`$k-1$`]` contains positive integers that represent the starting position in `var_clss` for the clause indexes of the variable v_k. The clause indexes in `var_clss` are encoded as integers, where the absolute value of them represents the clause index itself, and the positive or negative sign states if the referred variable has positive or negative polarity, respectively. Figure 4 shows an example diagram of both structures.

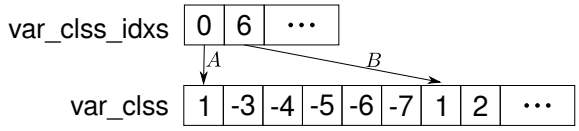

Fig. 4: Example diagram of the vector pair `var_clss` and `var_clss_idxs` for the formula $\mathcal{G}$ defined in equation 1 and encoded according to figure 5. The annotated arrows mark the first clause index of each variable.

Thanks to these new structures, the propagation loop only needs to iterate over all the clauses with a literal corresponding to a recently assigned variable. There can be multiple assigned variables that must be checked: the current decision variable, and all the variables that have been propagated when analysing the effects of the assignment of the decision variable.

Furthermore, since the polarity of a given variable literal is also known, only those clauses where the assignment did not satisfy (i.e. the polarity does not match the assigned truth value, or $\delta(l) \neq \lambda(\phi(l))$) need to be checked, or otherwise the clauses would be already satisfied.

3.3.2 Memory management

One problem with the data structure `vars`, and the newly added `clss_sat`, is that as now both conform the execution state of the program, both need d levels of depth to make the backtracking step possible. This is not really efficient, as the spatial complexities of these structures are $O(nd)$ and $O(md)$, respectively (where n is the number of variables and m the number of clauses). In the worst case, when $d = n$, the spatial complexities are $O(n^2)$ and $O(mn)$.

To use memory more efficiently, we encode the changes done at each level of the decision tree on the `vars` and `clss_sat` data structures, using a stack of logs. Each log entry contains all the changes done on the variable/clause state for a given decision level. The backtracking step implemented in a function called `backtrack` retrieves in LIFO order (i.e. last in first out, or most recently added first) the state changes and, therefore, reverse them.

This modification of the program yields a significant amount of saved memory, as now the space complexity for each of the logs of changes scales linearly with the number of variables and clauses (i.e. $O(n)$ and $O(m)$).

3.4 Parallel version

Lastly, with a more efficient solver in place, it was time to parallelize its execution. The algorithm corresponds to a divide-and-conquer pattern, which matches very well a task-level type of parallelism. We used the OpenMP framework [12] to implement the parallel code.

Given a number k, a problem of n variables is splited into 2^k tasks, each of which solves a sub-problem where k variables are already assigned to a different permuted state, leaving $n-k$ variables left to be assigned. This division of the search space and the creation of each task is implemented in a function named `partition_solve`. Since tasks need not share any information between them, this function is also accountable for the memory allocation of all the data structures that form the execution state of the program (i.e. `vars`, `clss_sat` and their respective stack logs). Therefore, all the local data structures are declared as thread private, upon the execution of each task. When all this is done, the function just relegates execution to the main DPLL procedure `solve` and deallocates the memory used.

Conceptually, the divide-and-conquer structure, paired with the usage of task parallelism, creates 2^k tasks that will be executed concurrently, with some number of them being executed in parallel (depending on the total number tasks and the computation resources at hand). Leaving the parameter k unfixed is useful, since not all tasks equal to the same amount of work (i.e. some will be easier or more difficult than others) and setting a specific value for k allows for some fine tuning of the desired number of spawned tasks for a specific problem. Practically speaking, it allows the adjusting of the overhead cost, intrinsic to the creation of more tasks, versus the better load balancing capability among the available threads that smaller problems may provide. This results in the ability to decide on an empirically optimal number k, which yields the best execution time on an individualistic problem fashion.

4 RESULTS

This section topic is about the execution of some problems and the performance related results that where obtained when executing them with the previously described versions.

4.1 Experimental methodology

All of the solver versions have been written using the C programming language [13] and have been compiled with the GNU Compiler Collection (GCC) on its version 12.20. The performance related compiler flags are `-fshort-enums`, `-O3`, `-march=native` and `-fopenmp` for the parallel version. The CPU used for execution is an Intel i5-6600k

CPU @ 3.50 GHz with 4 physical cores and no multithreading.

The performance metrics in table 2 have been obtained with the Performance Analysis Tools for Linux `perf` by executing each compiled version with a set of three problems.

4.2 Workload

The problems used to test the various versions of the solver are instances extracted from the SATLIB collection of benchmark problems [14]. Table 1 contains the main characteristics for each of the problems.

In terms of the size in number of variables $|V|$, it can be seen that all problems are in the 150 to 300 range. Meanwhile, with respect to the number of clauses $|C|$, the problem `3blocks` is far larger than the other two, with 9000 clauses versus 600, approximately 14-15x more. The same can be said, for the number of literals, with roughly the same increase of 14x, making the three problems having roughly the same length of 3 literals per clause. This makes `3blocks` larger in terms of clause/literal dimensions.

On the other hand, with respect to the satisfying assignments, we have that `aim-200-3_4-yes1-1` has only one, `3blocks` 174 and `uf150-011` 160 million, a far greater number of solutions with respect to the other two.

Problem	$\|V\|$	$\|C\|$	$\|L\|$	SAT-assignments
aim-200-3_4-yes1-1	200	680	2038	1
3blocks	283	9,690	26,810	174
uf150-011	150	645	1935	160,528,182

TABLE 1: NUMBER OF VARIABLES, CLAUSES, TOTAL LITERALS AND SATISFYING ASSIGNMENTS FOR EACH OF THE PROBLEMS

4.3 Analysis

Table 2 shows a summary of multiple execution metrics for each of the proposed versions, and each of the problems listed in the Table 1.

Analysing these results, a performance improvement trend can be seen across each incremental version, for all the three problems. The optimized version, with respect to the baseline one, solves the same problems 15.12x, 51.26x and 39.47x times faster, being the best performance increase for the problem `3blocks`, the one that has more clauses and more literals (even though the later is directly proportional to the increase in the number of clauses).

This result is to be expected, since the main advantage of the optimized version is the reduction of processed clauses, directly reflected in the reduction of the number of executed instructions.

Analogously, the parallel version performs 3.08x, 3.64x and 2.77x times faster than the optimized sequential version. Theoretically, the maximum performance increase would be 4x (we use 4 cores, each executing one thread) assuming perfect load balancing of the tasks to the execution threads and no slowdown in the core frequency of the processor.

As discussed before, the k parameter comes into play in the load balancing task. A different k was used for each problem, selected manually based on empirical testing. Starting from $k = 2$, k was increased with increments of 1, up until a point where the execution time is higher than the previous execution.

With this in mind, it can be said that the parallel execution of `3blocks` comes very close to the optimum, with a 91% efficiency using $k = 5$ ($2^5 = 32$ tasks). However, for `uf150-011`, with a much larger $k = 11$, only a 2.77x improvement is achieved, which means that the efficiency lowers to 69.25%. The explanation is that, depending on the problem and the number of total generated tasks, the difference between the bigger computation tasks versus the smaller tasks changes, and some configurations of problem and k are more unbalanced than others.

On the other hand, it's worth noting that, overall, the number of executed instructions is practically the same for the optimized and parallel versions, with less than 1% increase. What's interesting is that, only for `uf150-011`, the parallel version is the one which has less instructions, which is odd, since, in theory, the execution of additional overhead for the parallel code makes the opposite be true.

Finally, another point of interest is the fact that, for `uf150-011`, both optimized and parallel versions have a relatively low time % for the unit propagation procedure in comparison with the baseline version for the same problem, and also with the executions for the other two problems. A profiling if these two cases further revealed that most of the execution time was instead being wasted on the checking for a satisfying solution, a process embedded in an auxiliary function `all_sat` that iteratively checks the contents of `clss_sat`). This was then determined to be related to the huge number of solutions of the problem with respect to the others, since for each satisfying assignment found, a check of all the clause states is consequently done (since no F truth exists to stop the execution). More evidence for this is the fact that the baseline version is unaffected because the checking step for a satisfying assignment is embedded in the unit propagation (since it iterates over all the clauses), while the other versions do not.

5 CONCLUSIONS

In conclusion, this work has been of great for learning the current state of the art and main applications of SAT solvers, the fundamental algorithm DPLL and its unit propagation process, and, in general, more deeply about SAT solving.

Some immediate future lines of work could be the improvement of the parallelization, focusing on a more clever approach to the load balancing of the work. Also, at some point, CDCL, dynamic heuristics or more efficient data structures could be implemented. Aside from functionality, more testing, with more problems and more resources at hand could be of interest to qualitatively improve the performance analysis.

ACKNOWLEDGMENTS

I want to thank my tutor Juan Carlos for all the help in the development of this project and also to my closest people for all the support.

Problem	Version	Time (s)	IPC	Instructions (G)	Rel. Speedup	`unit_prop` (time %)	`backtrack` (time %)
aim-200-3_4-yes1-1	Baseline	1276.25	2.73	13551.30	-	99.70	-
	Optimized	84.39	2.11	694.00	15.12x	81.94	6.51
	Parallel ($k = 6$)	27.37	1.71	698.93	3.08x	84.70	13.87
3blocks	Baseline	1724.51	3.45	23064.15	-	99.88	-
	Optimized	33.64	2.58	338.79	51.26x	86.89	5.02
	Parallel ($k = 5$)	9.25	2.43	341.57	3.64x	88.33	10.57
uf150-011	Baseline	316.95	3.18	3938.36	-	98.28	-
	Optimized	8.03	3.49	109.25	39.47x	10.90	1.03
	Parallel ($k = 11$)	2.90	3.46	106.39	2.77x	12.06	2.87

TABLE 2: EXECUTION TIME, IPC, INSTRUCTIONS, RELATIVE SPEEDUP (WITH RESPECT TO THE EXECUTION OF THE SAME PROBLEM ABOVE IT) AND % OF THE TOTAL TIME EXECUTING `UNIT_PROP` AND `BACKTRACK` FUNCTIONS, RESPECTIVELY, FOR EACH OF PROPOSED SOLVER VERSIONS, GROUPED BY PROBLEM.

REFERENCES

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158. [Online]. Available: https://doi.org/10.1145/800157.805047

[2] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. NLD: IOS Press, 2009.

[3] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, p. 201–215, jul 1960. [Online]. Available: https://doi.org/10.1145/321033.321034

[4] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, p. 394–397, jul 1962. [Online]. Available: https://doi.org/10.1145/368273.368557

[5] P. Beame, H. Kautz, and A. Sabharwal, "Towards understanding and harnessing the potential of clause learning," *Journal of artificial intelligence research*, vol. 22, pp. 319–351, 2004.

[6] J. Marques Silva and K. Sakallah, "Grasp-a new search algorithm for satisfiability," in *Proceedings of International Conference on Computer Aided Design*, 1996, pp. 220–227.

[7] Z. Fu, Y. Marhajan, and S. Malik, "Zchaff sat solver," *Princeton University. Princeton, NJ*, vol. 8544, 2004.

[8] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 53, no. 2005, pp. 1–2, 2005.

[9] G. Audemard and L. Simon, "Glucose: a solver that predicts learnt clauses quality," *SAT Competition*, pp. 7–8, 2009.

[10] S. O. Comitee, "The international sat competition web page." [Online]. Available: http://satcompetition.org/

[11] H. H. Hoos *et al.*, "On the run-time behaviour of stochastic local search algorithms for sat," in *AAAI/IAAI*, 1999, pp. 661–666.

[12] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[13] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[14] H. H. Hoos, "Satlib - benchmark problems." [Online]. Available: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

APPENDIX

A.1 DIMACS CNF file format

The file format utilized to encode all the problems is known as DIMACS CNF.

It's a plain text format that starts with a header `p cnf <n> <m>`, where `n` is the number of variables and `m` the number of clauses of the formula.

Then, since it represents a formula in CNF, it encodes the literals of each clause as a spaced string of integers, where the minus sign represents a negation, delimiting the boundaries of each clause with a 0 and a newline at the end.

Additionally, comments can be added as any line starting with the letter `c`.

```
p cnf 6 7
1 2 0
2 3 0
-1 -4 5 0
-1 4 6 0
-1 -5 6 0
-1 4 -6 0
-1 -5 -6 0
```

Fig. 5: An example encoding of the equation 1, in the DIMACS CNF file format, where where literals $A = 1$ and $\neg A = -1$, $B = 2$ and $\neg B = -2$, ...

A.2 Code

The code for all the presented All-SAT solver can be found at `https://github.com/saul44203/sat-solver`.