TÉCNICO
LISBOA

Parallel SAT Solver

Ricardo de Sousa Marques

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Júri

Presidente:	Prof. Nuno Cavaco Gomes Horta
Orientador:	Prof. Paulo Ferreira Godinho Flores
Co-orientador:	Prof. Luís Miguel Teixeira D'Ávila Pinto da Silveira
Vogais:	Prof. Vasco Miguel Gomes Nunes Manquinho
	Prof. Nuno Filipe Valentim Roma

Outubro de 2013

Acknowledgments

First of all, I would like to thank all the ParSat group, specially Professors Luís Silveira and Paulo Flores for their guidance and support through all my academic course. They have always guided me with their knowledge whenever I would need some advice. Without the countless brainstorms and discussions this dissertation would never had succeeded. Their dedication to research and education will always be an example for me to follow.

I also would like to thank my family for their support, specially to my parents, for always being so close and being so supportive in all moments. It would be impossible to me to expresse my gratitude merely in words.

To all my friends and my girlfriend, Sofia, who make me feel that life is much more than work and have always stood by my side, through the best and worst moments.

Abstract

Many practical problems in multiple fields can be converted to a Boolean Satisfiability (SAT) problem, or a sequence of SAT problems, such that their solution immediately implies a solution to the original problem. Despite the enormous progress achieved over the last decade in the development of SAT solvers, there is strong demand for higher algorithm efficiency to solve harder and larger problems. However, algorithmic developments has slowed down somewhat, which is at odds with such a trend. Fortunately, the widespread availability of multi-core, shared memory parallel environments provides a renewed opportunity for such improvements.

In this dissertation we present our results on improving the effectiveness of standard SAT solvers on such architectures, through two different parallel approaches. In the first approach, multiple instances of the same basic solver using different heuristic strategies, compete to find a solution, each performing a different search-space exploration and problem analysis, but still sharing information and cooperating towards the solution of a given problem. In the second approach, we partition the problem in several sub-problems, using clustering techniques. Then, multiple instances cooperatively solve these smaller subsets of the original problem to find a solution for the whole problem.

Results from the application of our methodology to known problems from SAT competitions show improvements over the state of the art and yield the promise of further advances.

Keywords

Boolean Satisfiability (SAT), Parallel Search, Multi-core, Clause Sharing, Portfolio Solver, Clustering Algorithm

Resumo

Muitos problemas de natureza diversa podem ser convertidas num problema de satisfação booleana (SAT), ou numa sequência de problemas SAT, de tal forma que a solução para destes dá-nos a solução do problema original. Apesar do enorme progresso alcançado na última década no desenvolvimento de SAT *solvers*, existe a necessidade de aumentar a eficiência algorítmica, no sentido de resolver problemas cada vez maiores e mais difíceis. Contudo, estes desenvolvimentos abrandaram, contrastando com esta necessidade. Felizmente, o rápido crescimento de ambientes de computação paralela, nomeadamente multi-core, providencia-nos uma excelente oportunidade para esses progressos.

Nesta dissertação são apresentados os resultados dessa abordagem ao problema de SAT, sendo apresentados dois *solvers* que utilizam técnicas distintas para a resolução do problema: Na primeira estratégia, múltiplas *threads* executam o mesmo algoritmo, diferenciando entre si nas suas configurações, explorando desta forma o espaço de procura de forma distinta. Adicionalmente, as diferentes *threads* partilham informação entre si, acelerando a resolução do problema. Na segunda estratégia é efectuada uma partição do problema em vários subproblemas, usando técnicas de *clustering*. Seguidamente, diversas instâncias resolvem estes subproblemas e cooperam entre si até alcançarem uma solução.

Os resultados da aplicação dos algoritmos implementados a problemas retirados de competições de SAT demonstram algum progresso sobre os melhores algoritmos sequenciais e justificam uma investigação mais profunda sobre este tema.

Palavras Chave

Satisfação booleana (SAT), Procura Paralela, Multi-core, Partilha de Cláusulas, Portfolio, Algoritmos de Partição

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

Contents

A motivating problem

There are three balls in a box: a red ball, a green ball and a blue ball. Suppose we have take some of them from the box with some conditions: (i) At least one of the red ball and green ball should be selected, (ii) either the green ball won't be selected or the blue ball will be selected, and (iii) either the red ball won't be selected or the blue ball will be selected.

How can we determine which balls we can take from the box? This logic problem can be written as a propositional formula.

Let's consider the boolean variables R, G and B, which represent the red, green and blue balls. If these variables take the value 1 (*true*) this means that they will be picked. Otherwise, they will take the value 0 (*false*) and will not be selected.

The following clauses encode the constraints mentioned:

- $(R \vee G)$: At least one of the red ball and green ball should be selected
- $(\neg G \vee B)$: Either the green ball won't be selected or the blue ball will be selected
- $(\neg R \vee B)$: Either the red ball won't be selected or the blue ball will be selected.

With this constraints, is it possible to find a set of assignments in such a way that we satisfy all clauses? The process of finding this assignments is called Boolean Satisfiability (SAT).

What is SAT?

Given a Boolean formula, the problem of determining whether there exists a variable assignment that makes the formula evaluate to true is called the satisfiability problem. If the formula is limited to only contain logic operations **and**, **or** and **not**, then the formula is said to be a propositional Boolean formula. Determining the satisfiability of a propositional Boolean formula is called the Boolean Satisfiability Problem (SAT).

One possible set assignments that would satisfy the above case would be $R = 1$, $G = 0$ and $B = 1$. This means that taking the red and the blue ball from the box would satisfy all constraints.

Why is SAT important?

The Boolean Satisfiability Problem (SAT) is one of the most important and extensively studied problems in computer science. Boolean Satisfiability has been the first problem proven to be NP-Complete [8], Nowadays, due to the enourmous advances in computational SAT solvers, the SAT problem is of practical importance in a wide range of disciplines, such as planning Automatic Test Patten Generation (ATPG) [22], combinational circuit equivalence checking [13], sequential property checking [29], microprocessor verification [34], model checking [5], redundancy removal [20] and timing analysis [30], amongst others.

1.1 Motivation

The widespread use of SAT is the result of SAT solvers being so effective in practice. Many real-life, industrial problems, with hundreds of thousands of variables and millions of clauses, are

routinely solved within a few minutes by state of the art SAT solvers.

This added capability of state of the art SAT solvers, however, is continuously challenged by emerging applications which bring to the fold problems and systems of increasing size and complexity. As a consequence, in spite of the remarkable gains we have seen in the area, there are still many problems that remain very challenging and unsolved by even the best solvers. Perhaps the main cause for this situation is that the large, steady algorithmic improvements that were made available in the last decade with the introduction of powerful techniques such as non-chronological backtracking, restarts, improvements in decision heuristics, etc, seem to have slowed down. Improvements to SAT solvers nowadays appear to be more incremental, sometimes problem-related. Faced with this situation, researchers have started to look elsewhere for ways to continue improving the efficiency of SAT solvers and the widespread availability of parallel computing platforms provided renewed opportunities to achieve this goal.

The generalization of multi-core processors, as well as the availability of fairly standard clustering software, provided access for the common user to parallel computing environments and has opened up new opportunities for improvement in many areas. In this context, many parallel SAT solvers have been proposed and SAT competitions now routinely include a parallel track. The main goal of parallel SAT solvers is to be able to solve problems faster. Several metrics can be used to quantify the resulting improvements, including speedup and efficiency. The basic obstacles to improved efficiency are generally the same ones as encountered by parallel implementations of other problems and domains, namely load balancing and robustness, i.e. the ability to sustain similar efficiency over a large range of problems. An enticing feature of parallelization in search-based problems is that super-linear speedups are achievable if one is lucky to search the right region of the search space. Often, one is satisfied with the ability to demonstrate speedup by solving problems faster and doing so in a robust manner over a large range of problems. Combining such robustness with the elusive possibility of large gains is clearly a goal worth pursuing.

1.2 Objectives

The goal in this thesis was to research and develop techniques to speedup the solution of SAT problems, using distributed SAT algorithms, running on parallel computing environments.

Targeting this goal two parallel multi-core SAT solvers were developed: pmcSAT and clusterSAT, which can be readily be described as follows:

- **pmcSAT** launches multiple instances of the same solver, with different parameter configurations, which cooperate to a certain degree by sharing relevant information when searching for a solution. This approach is known as a portfolio approach.

- **textscclusterSAT** uses graph partitioning libraries to split the problem into smaller subproblems and launches multiple tasks to solve these subproblems and cooperate to find a

valid solution.

A more detailed description of these solvers and the respective results is shown in the next chapters.

1.3 Main contributions

The main contributions of the work developed through this thesis correspond to the developed parallel solvers:

- **pmcSAT**, although we do not consider it to be a novel contribution, has shown remarkable results at the SAT Competition 2013 [2], being awarded with a bronze medal on one of the Competition's Parallel Tracks. This solver has also been presented in Florida, USA, at the FLAIRS-26 Conference [24].
- **clusterSAT**, despite the fact that has not shown remarkable results, has helped to develop an approach still in its infancy and could be enhancer for future successful parallel SAT solvers

1.4 Dissertation outline

The remainder of this dissertation is organized as follows: **Chapter 2** overviews satisfiability algorithms. We introduce the Davis-Putnam (DP) resolution-based algorithm, followed by the Davis-Logemann-Loveland (DLL) backtrack search algorithm. Afterwards, a detailed description the fundamental techniques of SAT solving is given, namely non-chronological backtracking, decision heuristics, boolean constraint propagation and backtracks.

Chapter 3 reviews and discusses existing parallel approaches, as well the state of the art parallel SAT solvers. A brief description of the most common challenges found in parallel SAT solving is also given.

Chapter 4 describes the first solver developed through this thesis, pmcSAT. The implementation details are given, as well as a description of the techniques used, followed by the experimental results. These results include a comparison with the sequential implementation and also the SAT Competition 2013 results.

Chapter 5 describes the second solver developed, clusterSAT. This chapter follows the same structure as Chapter 4, with the implementation details followed by experimental results.

Finally, in **Chapter 6** we conclude the dissertation and suggest future research work.

The work developed on this thesis was partially supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under the project "ParSat: Parallel Satisfiability Algorithms and its Applications" (PTDC/EIA-EIA/103532/2008)".

2

SAT Overview

Contents

This chapter provides an overview of SAT solving algorithms. Section 2.1 introduces the basic concepts used in SAT solvers. In the following sections the definitions used through this thesis are provided. Moreover, other features of a competitive SAT Solver are described, namely decision heuristics, propagation, non-chronological backtrack and restarts.

2.1 Basic Concepts of SAT Solving

- A **conjunctive normal form (CNF)** of a formula φ consists in a conjunction (logic **and**) of one or more clauses.
- A **clause** ω is a disjunction (logic **or**) of one or more literals.
- A **literal** l is the occurrence of a boolean variable x_i in its *positive phase* (x_i) or in its *negative phase* ($\neg x_i$).

An example of a propositional formula in CNF is:

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \tag{2.1}$$

This formula contains two clauses: $\omega_1 = x_1 \vee x_2 \vee x_3$ and $\omega_2 = \neg x_2 \vee x_4$.

A literal is satisfied if its truth value is 1 and unsatisfied if its truth value is 0. A literal with no truth value is said to be unassigned. A clause is said to be satisfied if at least one of its literals is satisfied, and it is said to be unsatisfied if all of its literals are unsatisfied. A literal is also considered *pure* if its complement does not occur in the formula. A clause having a variable and its complement is called a *tautology* and is always satisfied, regardless the given assignment. A clause with no literals is an empty clause and is always unsatisfied.

Given a propositional Boolean formula, the Boolean Satisfiability Problem (SAT) consists in finding a set of variable assignments in such a way as to make the formula true (the problem is **SATISFIABLE**), or to determine whether no such assignments exist (**UNSATISFIABLE**). For instance, in the above equation, if we consider the set of assignments $\rho = \{x_1 = 1, x_2 = 0\}$ we can easily verify that the formula is *satisfiable*.

Currently almost all modern SAT solvers limit the input to be in CNF, due to the easy translation from real world applications into CNF instances. This form also can provide the user the option of adding more clauses to the problem, which can be useful to prune the search space.

2.2 Davis-Putnam (DP) Resolution-Based Algorithm

Because of the practical importance of Boolean Satisfiability, significant research effort has been spent on developing efficient SAT solving procedures for practical purposes. The original algorithm for SAT solving was proposed by Davis and Putnam [10] in 1960. This algorithm is

Algorithm 2.1 Davis-Putnam algorithm

```
1: while TRUE do
2:     if BCP(φ) == CONFLICT then
3:         return UNSATISFIABLE;
4:     end if
5:     Pure_Literal_Rule(φ);
6:     if All_Clauses_Satisfied(φ) then
7:         return SATISFIABLE;
8:     end if
9:     x = Select_Variable(φ);
10:    Eliminate_Variable(φ, x);
11: end while
```

resolution-based. Resolution is an inference rule for propositional logic and can be regarded as a general technique for deriving new clauses.

Given two clauses, $\omega_1 = (z \vee x_1 \vee ... \vee x_m)$ and $\omega_2 = (\neg z \vee y_1 \vee ... \vee y_n)$ where x_i and y_j are different sets of literals, we can deduce the following disjunction: $\omega_3 = (x_1 \vee ... \vee x_m \vee y_1 \vee ... \vee y_n)$, where we can remove z and $\neg z$. Repeated iterations of this method will reduce the formula to a set of clauses having only pure literals, in case of the problem being satisfiable, or otherwise will lead to an empty clause, in the case of being unsatisfiable.

The DP algorithm also uses the following techniques:

- **Pure literal rule** - Given a formula φ that contains pure literals, a set of assignments ρ could be made in such a way that those literals are satisfied. This way, all the clauses that contain these pure literals are satisfied and can be removed from the problem without any restrictions to the original formula.
- **Unit clause rule** - Given a clause ω in which all its literals are unsatisfied but one, the remaining literal, called **unit literal**, has to be satisfied in order to satisfy ω. This clause is also named as **unit clause**.
- **Boolean Constraint Propagation** - Process of analysing the formula after performing an assignment and, if needed, application the unit clause rule.

These techniques are described in the following example:

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_3 \vee x_4) \tag{2.2}$$

Given the above formula, if we consider the assignments $\rho = \{x_1 = 0, x_2 = 0\}$, in order to satisfy the first clause the assignment $x_3 = 1$ has to made and consequently $x_4 = 1$ in order to satisfy the second clause.

The pseudo-code for the Davis-Putnam algorithm is given in Algorithm 2.1. Besides applying BCP and the pure literal rule, the purpose is to iteratively applying the resolution process by eliminating variables until satisfiability or unsatisfiability can be determined. A formula is declared

to be satisfied when it contains only satisfied clauses or pure literals. A formula is declared to be unsatisfied whenever a conflict is reached.

This process, however, in each iteration of the algorithm generates a sub-problem with one fewer variable but probably with more clauses. The algorithm requires exponential space to solve an instance. Therefore, alternative techniques to solve to this problem were developed and are described in the following sections.

2.3 Davis-Logemann-Loveland (DLL) Search-Based Algorithm

The DP algorithm quickly leads to memory overload problems, so in 1962 Davis, Logemann and Loveland proposed a search-based algorithm, also known as DLL or DPLL [9]. This algorithm is the most widely studied algorithm for SAT solving. Unlike the DP algorithm, the memory requirement for this algorithm is not exponential, therefore SAT solvers that use this approach can solve very large formulas without memory overflow.

The DLL algorithm is a branch and search algorithm. The search space is often presented in binary trees, such as in figure 2.1.

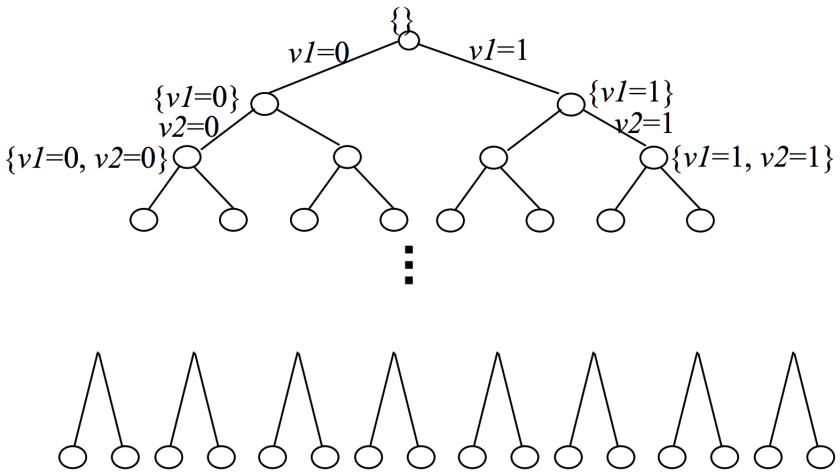

Figure 2.1: Example of a decision tree

Each node in the tree represents a variable assignment. A *decision level* is associated in each assignment to indicate its depth at the decision tree. For each new decision assignment, the decision level increments by 1. After each assignment, propagation is executed and each implied assignment is associated with a *reason*, i.e., the unit clause that implied the assignment.

Algorithm 2.2 describes a generic DLL algorithm. Given a formula φ, variables are assigned iteratively through the DECIDE_NEXT_BRANCH function. Following this assignment, implied assignments are identified in the DEDUCE function. If this implications lead to a conflict, a DIAGNOSE is performed, which identifies the decision level that originally generated the conflict. The process

Algorithm 2.2 Davis-Logemann-Loveland algorithm

```
1: while TRUE do
2:     v = DECIDE_NEXT_BRANCH(φ);
3:     if !v then
4:         return SATISFIABLE;
5:     end if
6:     if DEDUCE(φ, v) == CONFLICT then
7:         β = DIAGNOSE(φ);
8:         if β == 0 then
9:             return UNSATISFIABLE;
10:        else
11:            BACKTRACK(φ, β);
12:        end if
13:    end if
14: end while
```

of undoing all decisions and corresponding implications to that level is called BACKTRACK. If this backtrack level is at the top of the decision tree ($\beta = 0$) then the problem is **UNSATISFIABLE**. The problem is considered to be **SATISFIABLE** when all variables are assigned and no conflicts are detected, i.e., all clauses are satisfied.

Different SAT solvers based on DLL differ mainly on the implementation and techniques used of the various functions in the top level algorithm. In the next sections, a more detailed description of these functions is done.

2.3.1 Backtrack

Whenever a SAT solver reaches a conflict, the solver needs to undo the assignments and return to a consistent state. This process is named *backtrack*.

Original DLL algorithms use **chronological backtracking**. This method consists in keeping a flag for each variable in the search tree that marks if both assignments have been tried. (i.e., flipped). When a conflict occurs, the solver undoes the last decision that is not been flipped and resumes the search.

This method, however, is often not effective in pruning the search space. State-of-the-art SAT Solvers perform a deeper conflict analysis and build a conflict clause that determines the origin of the conflict. This clause is learned (i.e., added to the problem) and backtracking is then performed, possibly undoing several decisions, instead of blindly backtracking chronologically without adding any additional information to the problem. This process is named **non-chronological backtracking**.

Figure 2.2 describes an example of chronological and non-chronological backtracking. As we can observe, a conflict is detected on the assignment of the variable x_4. While chronological backtracking simply undoes the assignment of $x_3 = 0$ and resumes the search process with $x_3 = 1$, the conflict analysis from non-chronological backtracking detects that the conflict reason is in the above level, where $x_2 = 0$ has been assigned and resumes the search process in a region of the search

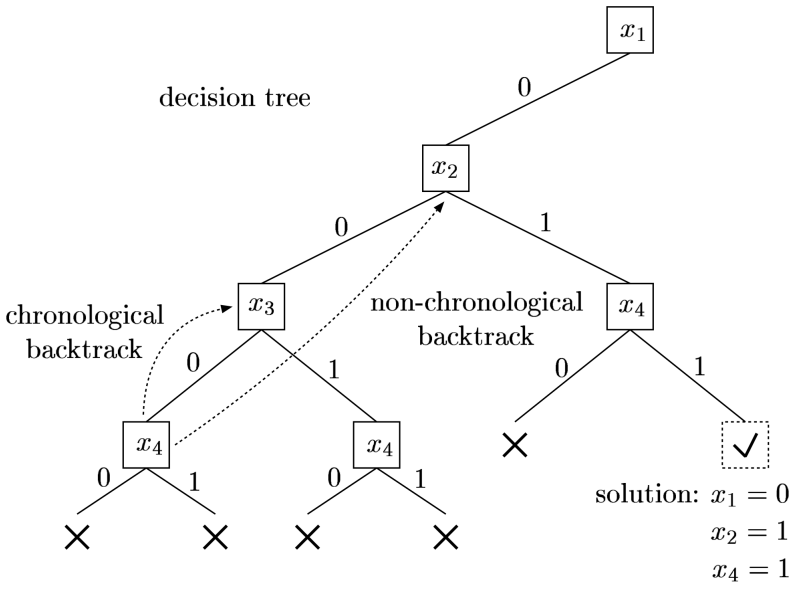

Figure 2.2: Chronological and Non-Chronological Backtracking

space that contains the solution to the problem. On the other hand, chronological backtracking could waste a significant amount of time exploring a region of the search space where no solution can be found.

2.3.2 Decision Heuristics

Decision heuristics play a key role in the efficiency of SAT algorithms, since they determine which areas of the search space get explored first. A well chosen sequence of decisions may instantly yield a solution, while a poorly chosen one may require the entire search space to be explored before a solution is reached.

Over the years, many different branching heuristics have been proposed. The first successful heuristics, such as s Bohm's Heuristic [6], Maximum Occurrences on Minimum sized clauses (MOM) [12], and Jeroslow-Wang [16] can be considered greedy algorithms that try to generate the largest number of implications, or to satisfy the most clauses.

Bohm's Heuristic gives preference to variables that occur more often in smallest clauses. The purpose is to assign the variables that satisfies most clauses (when assigned *true*), and to reduce the size of small clauses, generating more implications (when assigned *false*). At each iteration of the algorithm, the Bohm's heuristic selects the variable that maximizes the vector $< H_1(x), H_2(x), ..., H_n(x) >$, where $H_i(x)$ is computed by the following equation:

$$H_i(x) = \alpha * max(h_i(x), h_i(\neg x)) + \beta * min(h_i(x), h_i(\neg x)) \tag{2.3}$$

where h_i is the number of unresolved clauses with i literals containing x. α and β are parameters, originally suggested as $\alpha = 1$, $\beta = 2$. The major issue of this heuristic is the fact that it is very expensive to compute these vectors at every decision level.

The **MOM's heuristic** gives preference to variables that occur in the smallest clauses, but variables are preferred if they simultaneously maximize their number of positive and negative literals in the smallest clauses. Let $f^*(l)$ be the number of clauses of the smallest length containing the literal l. The variable that maximizes the following formula is chosen:

$$[f^*(x) + f^*(\neg x)] * 2^k + f^*(x) * f^*(\neg x) \tag{2.4}$$

Analyzing the formula we can easily verify that preference is given to variables that belong to a large number of clauses which have the smallest length. Within this set, preference is given to variables that have a similiar count of both x and $\neg x$, assuming that k is a very large parameter.

The **Jeroslow-Wang Heuristics** compute the weight of a literal l according to the following formula:

$$J(l) = \sum_{l \in \omega \wedge \omega \in \phi} 2^{-|\omega|} \tag{2.5}$$

where in each clause ω where l appears, a value of $2^{-|\omega|}$ is added to its score, where $-|\omega|$ is the number of literals in ω.

The one-sided version of this heuristic (JW-OS) selects the assignment that satisfies the literal with the highest value of $J(l)$. The two-sided version (JW-TS) identifies the variable x with the largest sum of $(J(x) + J(\neg x))$ and assigns x true in case of $J(x) \geq J(\neg x)$, otherwise assigns false.

Literal count heuristics count the number of unresolved clauses in which a given variable x appears as a positive literal, C_P, and as a negative literal, C_N. From combining these values, different heuristics were created:

- **Dynamic Largest Combined Sum (DLCS)** - This heuristic selects the variable with highest value of $C_P + C_N$, and assigns it true when $C_P \geq C_N$, and false otherwise.
- **Dynamic Largest Individual Sum (DLIS)** - This heuristic selects the variable with highest value of C_P or C_N, and assigns it true when $C_P \geq C_N$, and false otherwise.
- **Random Dynamic Largest Individual Sum (RDLIS)** - A variation of DLIS, which consists in choosing a random value for the selected variable, instead of comparing C_P with C_N. The random selection of the value to assign is in general a good compromise to prevent making too many bad decisions for a few specfic instances.

These heuristics, in spite of being dynamic, are *state-dependent*, i.e., each time the heuristic function is called, the counts for all the free variables need to be recalculated. This often introduces a large overhead to the SAT solver. A heuristic that is independent of the search-state was then proposed by CHAFF [26]: **Variable State Independent Decaying Sum (VSIDS)**.

In VSIDS the variables are ordered based on their activity. Because modern SAT solvers use learning, additional clauses are added to the clause database as the search progresses. VSIDS

increases the score of a literal by a constant whenever an added clause contains the literal. Additionally, as the search advances a periodical division of all scores by a number is performed. Therefore, this heuristic gives preference to variables that occur in more recent conflicts. VSIDS will choose the free variable that will have highest score.

VSIDS is a *state-independent* heuristic, i.e., the scoring does not depend from the variable assignment and, unlike the previous heuristics, takes only a small percentage of the processing time of a SAT Solver. The purpose of VSIDS, and similar activity based heuristics, is to avoid scattering the search, by directing it to the most constrained parts of the formula. These techniques are particularly effective when dealing with large problems.

2.3.3 Boolean Constraint Propagation

As mentioned previously in this chapter, Boolean Constraint Propagation is the process of analysing the formula after performing an assignment and, if needed, perform assignments to unit literals in order to satisfy unit clauses.

Since any SAT algorithm relies extensively on accessing and manipulating large amounts of information, its data structures are of paramount importance for its overall performance. In order to efficiently implement the BCP procedure, a data structure should allow us to find fast unsatisfied and unit clauses after each variable assignment. It should also allow us to make as few as possible operations to maintain the data structures consistent after backtracking.

The GRASP algorithm [31] uses a **counter-based** method. In this approach, two counters are used for each clause, one for the true assignments count and other for the false assignments. It also stores the total number of literals of the clause. Each variable also records a list of all clauses where it appears. Whenever a variable is assigned, all clauses that contain that variable update their counters. Three possible situations could result from an assignment:

- If the number of false assignments equals the total number of literals, then the clause is a conflict clause and further backtrack needs to be done.
- If the number of false assignments equals one less than the total number of literals in the clause and the true assignments count is 0, then the clause is a unit clause.
- Otherwise, the clause is neither conflicting nor unit and nothing needs to be done for the clause.

This approach, however, could be very inefficient. For instance, if a problem has n variables, m clauses and l literals per clause, then every time a variable is assigned, lm/n counters would be updated. The same number of counters would have to be updated every time that a backtrack is performed. Modern SAT solvers use learning as a mechanism and usually learned clauses have a high number of literals, which would turn this method of applying BCP very slow.

As we could verify, each variable keeps references to a potentially large number of clauses, that often increases as the search proceeds. This impacts negatively the amount of operations associated with an assignment. On top of this, most clauses associated with a variable assignment would not become unsatisfied or unit. The solutions to this issue are ***lazy data structures*** and are described bellow.

The first lazy data structure to apply BCP was proposed by Zhang and Stickel in the SATO SAT Solver [35], called the **Head/Tail data structure**. In this mechanism, each clause has two pointers associated, a *head*(H) and *tail*(T). Initially, the head points to the first literal of the clause and he tail to last literal. Both pointers move towards the center of the clause. Each variable x also records in which clauses it appears and it is referenced as a head/tail.

Each time a literal pointed to by either the head or tail reference is assigned, a new unassigned literal is searched for. When an unassigned variable is identified it becomes the new head, and a new reference is created and associated with the literal's variable. This guarantees that the H/T pointers are recovered in case of a backtrack. If a satisfied literal is identified, then the clause is satisfied and no further work is done. If no unassigned literal can be identified and the other reference is reached, then the clause is unit if the reference is unassigned, satisfied if the variable is assigned true, or unsatisfied if the reference is assigned false.

When a backtrack is performed, the current H/T references are discarded and the previous ones become activated.

Inspired by this data structure, the CHAFF SAT solver [26] proposed a new data structure, the **Watched Literals**. This method solves some issues of the H/T approach. It uses a similar structure, associating two references to each clause. However, unlike in H/T, these references can move in any direction. This approach has the drawback of only identifying a unit or unsatisfied clause when after transversing all the literals. On the other hand, the major advantage of this method is the fact that when backtrack is performed no references need to be updated.

Figure 2.3 illustrates these two methods of applying propagation.

2.3.4 Restarts

SAT algorithms can exhibit a large variability in the time required to solve any particular problem instance. Indeed, huge performance differences can be observed when using different decision heuristics. This behavior was studied by [14], where it was observed that the runtime distributions for backtrack search SAT algorithms are characterized by heavy tails. Heavy tail behavior implies that, most often, the search can get stuck in a region of the search space.

The introduction of restarts [19] was proposed as a method of minimizing the effects of this problem. The restart strategy consists of defining a threshold value in the number of backtracks, and aborting a given run and starting a new run whenever that threshold value is reached. Randomization must also be incorporated into the decision heuristics, to avoid the same sequence of

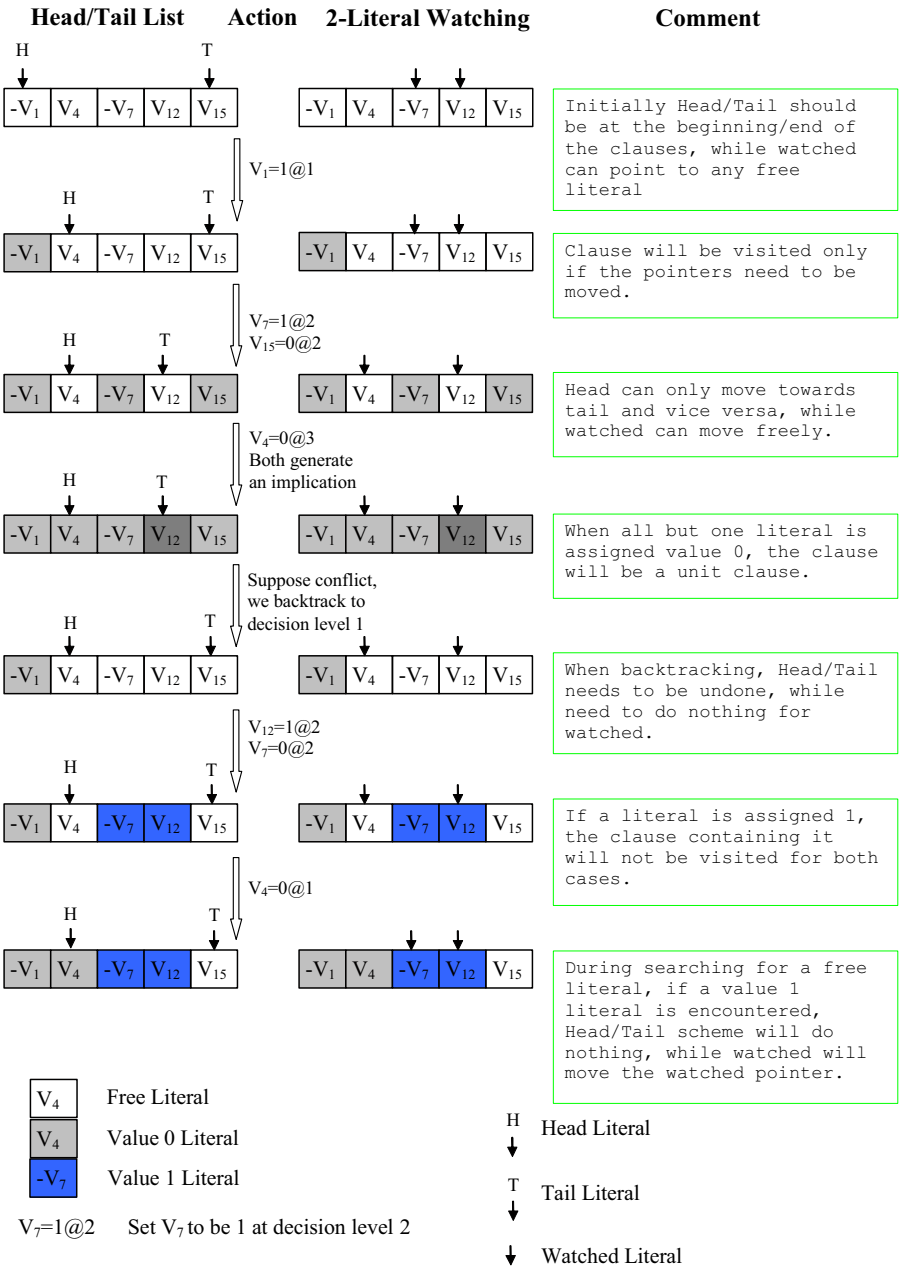

Figure 2.3: Head/Tail list scheme and Watched Literals scheme descriptions

decisions to be repeated on every run. In order to preserve the completeness of the algorithm, the backtrack threshold value must be increased after every restart, thus enabling the entire search space to be explored, after a certain number of restarts. Restarts and activity-based decision heuristics are complementary, since the first one moves the search to a new region of the search space, while the second one enables the search to be focused in that new region.

2.4 Modern CDCL SAT Solvers

Current state-of-the-art sequential SAT solvers are mostly **Conflict-Driven Clause Learning (CDCL)**. Algorithm 2.3 shows the standard organization of a CDCL SAT solver first proposed by Marques-Silva and Sakallah [31].

Algorithm 2.3 CDCL SAT Solver

```
1:  limit = c;
2:  while TRUE do
3:      while #conflicts < limit do
4:          lit = DECIDE();
5:          if !lit then
6:              return SATISFIABLE;
7:          end if
8:          if !BCP(lit) then
9:              cl = CONFLICT_ANALYSIS();
10:             if !cl then
11:                 return UNSATISFIABLE;
12:             end if
13:             #conflicts++;
14:         end if
15:     end while
16:     UNDO_DECISIONS();
17:     INCREASE(limit);
18: end while
```

This approach is DLL-based and uses the techniques described in the previous subsections. Some of the fastest SAT solvers that use these methods are briefly described in the following:

- **MINISAT** [11] is a minimalistic, open-source SAT solver developed by Niklas Eén and Niklas Sörensson. MINISAT is small, well-documented and easy to modify, making it an ideal starting point for adapting SAT based techniques to domain specific problems. It was also the winner of all the industrial categories of the SAT 2005, 2006 and 2008 competitions. MINISAT 2.2.0 has been the support solver for the parallel solvers developed through this thesis.

- **GLUCOSE** [1] is based on a different scoring scheme for the clause learning mechanism. The solver's name is a contraction of the concept of "glue clauses", a particular kind of clauses that glucose detects and preserves during search. GLUCOSE is heavily based on MINISAT and was the winner of many categories of the latest SAT competitions.

- **PICOSAT** [3] is a SAT solver, developed by Armin Biere, that uses some of the MINISAT techniques. It was the SAT 2007 competition winner and it is the base solver for the state-of-the-art solver LINGELING.

- **LINGELING** [4] is a SAT solver, based on PICOSAT and PRECOSAT. All of these solvers have been developed by Armin Biere. LINGELING has been one of the most successful in the latest SAT Competitions, among GLUCOSE and MINISAT.

2.5 Conclusions

This chapter gives an overview of the techniques used in sequential SAT solving. Some basic concepts commonly used in SAT were introduced. Different techniques for SAT solving were described. In particular, the most widely used SAT solving algorithm, namely the Davis Logemann Loveland (DLL) algorithm, was discussed in detail.

However, these techniques are applied only to purely sequential SAT solvers. The next chapter explains why should we move to **parallel SAT Solving**, the main challenges and the existing approaches.

3

Parallel SAT

Contents

State of the art sequential algorithms have revealed minor improvements over the recent years, while SAT is applied to larger and more ambitious problems which cannot be solved in reasonable time. The demand for more computation power led to single-core architecture alternatives, due to termal issues. These options are parallel architectures, such as multi-core processors, GPU's or clusters and grids. In the past years, parallel SAT Solving has been the focus of research in SAT Solving, since parallel SAT Solvers often provide faster resolution of SAT instances. In this section, we describe the most successful approaches in parallel SAT solving, as well the most common challenges.

3.1 Parallel approaches to SAT solving

There are two main approaches in parallel SAT solving. On one hand we have **cooperative solvers**, which use a set of constraints, also named guiding paths, to split the original search space. On the other hand we have **competitive solvers**, or **portfolios**, in which different computing units solve the same SAT instance, using different search mechanisms.

3.1.1 Cooperative Solvers

Search-space splitting

As mentioned above, these algorithms perform a search-space splitting into disjoint subspaces to be explored in parallel. The most common method to perform this is using guiding paths [36]. These guiding paths are the list of variables assigned until the current state of the search process. Each variable has a flag associated that records if both assignments, true or false, have been tried. A variable where both assignments have been tried is closed. On the other hand, a variable that still has values to be assigned is said to be open. Open variables represent disjoint spaces yet to be explored. Figure 3.1, illustrates an example of this approach.

However, the hardness of the different subspaces is frequently unbalanced, since some subspaces are easier to prove (un)satisfiability than others. Since the processing time for these subsets to be completed can't be predicted, a dynamic workload balance method has to be used in order to maintain balance between all units. This way we prevent threads quickly becoming idle, while others take too much time solving their subspace.

Most parallel search-splitting solvers use a master-slave approach. In this approach, the master controls a set of slave processes. The master process is responsible for keeping a workload balance between all slave processes. When a slave becomes idle, it sends a request to the master process for a new subspace. The master process selects the slave that has the shortest guiding path, splits its search and then provides to the idle slave a new search space.

However, the shortest guiding path, despite being the largest unexplored space of the search, may result in subspaces that could be quickly proven to be unsatisfiable. This could lead to a very inefficient CPU time management, since the master would be often interrupting the work of the

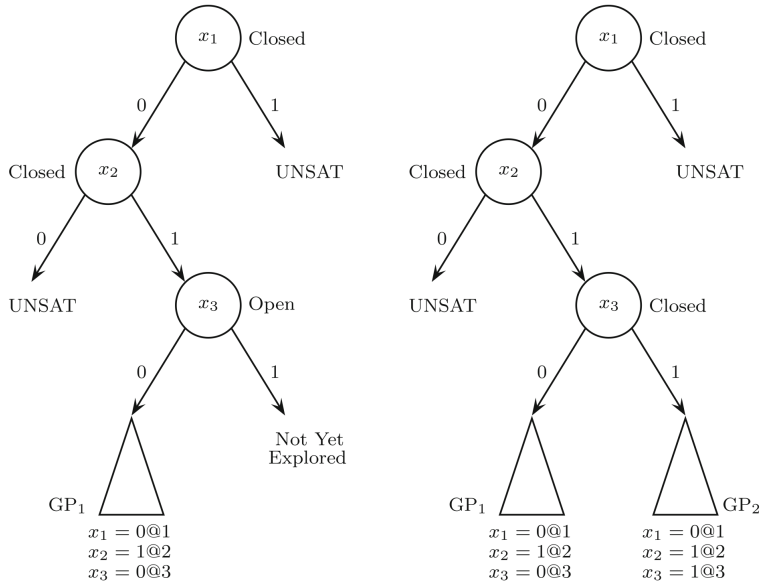

Figure 3.1: Search space splitting example. Figure obtained from [25]

slaves requesting for new subspaces. A solution that minimizes the idle time is to use a work pool that contains unexplored guiding paths. Slaves could periodically fill this queue in such a way that the master doesn't need to interrupt their search as often. This way, when a slave is idle, it first checks this queue if there are any guiding paths, only requesting the master for a new subspace when there are no remaining unexplored guiding paths in the work pool.

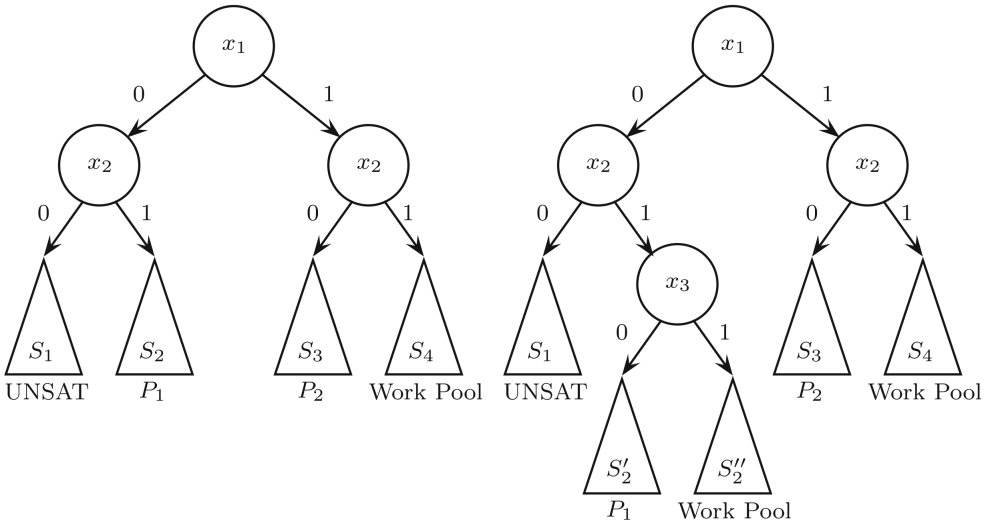

Figure 3.2: Dynamic work stealing example. Figure obtained from [25]

Parallel Solvers pMiniSat [7], MiraXT [23] and PaMira [28] use this approach. Figure 3.2 shows an example of this dynamic work stealing approach.

Problem splitting

Other approach to cooperative solvers is to split the original formula into several subformulas, instead of search-space splitting through guiding paths. This subformulas could be solved in parallel, although there is the need to have synchronism due to the shared variables between subproblems.

An example of a solver that uses this approach is JaCk-SAT [32]. This solver divides the variable set V in two subsets V_1, V_2 of about equal size. Let $P = (V, C)$ be the initial SAT problem, the decomposition step builds the two subproblems $P_1 = (V_1, C_1)$, $P_2 = (V_2, C_2)$ and a residual clause-set C_3. Each of the subsets C_1 and C_2 is composed of variables belonging only to V_1 and V_2, respectively. The remaining clause-set (C_3) is composed of variables that belong to both V_1 and V_2.

This decomposition results in searching independently for all the solutions of both subproblems P_1 and P_2, trying to join them to obtain global solutions of P, thus checking for satisfiability of the residual clauses C_3.

The major drawbacks of this method to be overcome are two-fold. First, the search for all the solutions of $n/2$ variables problems compared to the search for one solution of a n variables problem. Next, the additional Join-and-Check step that is polynomial in the number of solutions but may be exponential in terms of n.

3.1.2 Competitive Solvers

Competitive Solvers, or portfolios, approach the SAT problem using different strategies on the same instance in parallel. The performance of these algorithms is as good as the faster strategy that solves the problem. Additionally, if these algorithms cooperate between themselves by exchanging information like learned clauses it is possible to outperform the best strategy for a given problem. Usually, a parallel portfolio uses different parameters for each SAT algorithm. These can be different decision heuristic, restart or learning strategies, but they may also be completely different solvers, running in parallel.

ManySAT [15] was the first portfolio solver to use a multicore architecture. Table 3.1 describes the different strategies used in version 1.0, designed for 4 cores. For each core, the restart, decision heuristic, polarity and learning strategies are slightly different. Clause sharing is limited to a static size of 8 literals, i.e., all learned clauses that had 8 literals or less are shared between threads.

Even though parameter tuning could lead to complementary algorithms, it may also lead to strategies that perform similarly. Therefore, if we increase the number of computing units it may be difficult to achieve an efficient performance. A big challenge in portfolio solvers is to achieve scalability. The following versions of ManySAT are already suited for scalability, since these use diversification and intensification strategies. A set of master-slave pairs is used in such a way that the slave intensifies the search of the master by exploring its search space in a different way.

Strategies	Core 0	Core 1	Core 2	Core 3
Restart	Geometric $x_1 = 100$ $x_i = 1.5 \times x_{i-1}$	Dynamic (Fast) $x_1 = 100, x_2 = 100$ $x_i = f(y_{i-1}, y_i), i > 2$ if $y_{i-1} < y_i$ $f(y_{i-1}, y_i) =$ $\frac{\alpha}{y_i} \times \lvert cos(1 - \frac{y_{i-1}}{y_i}) \rvert$ else $f(y_{i-1}, y_i) =$ $\frac{\alpha}{y_i} \times \lvert cos(1 - \frac{y_i}{y_{i-1}}) \rvert$ $\alpha = 1200$	Arithmetic $x_1 = 16000$ $x_i = x_{i-1} + 16000$	Luby 512
Heuristic	VSIDS (3% rand.)	VSIDS (2% rand.)	VSIDS (2% rand.)	VSIDS (2% rand.)
Polarity	if $\#occ(l) > \#occ(\neg l)$ $l = true$ else $l = false$	Progress saving	false	Progress saving
Learning	CDCL (extended [1])	CDCL	CDCL	CDCL (extended [1])
Cl. sharing	size ≤ 8	size ≤ 8	size ≤ 8	size ≤ 8

Table 3.1: Different ManySAT 1.0 strategies

The results of ManySAT led to a win in the 2008 SAT Race [33] and with its predominance new portfolio solvers have been developed since then, including PLINGELING [4], SARTAGNAN [21] and PPFOLIO [27]. The first two run several algorithms that differ between themselves in the heuristics, while PPFOLIO is a compilation of different sequential state of the art solvers.

3.2 Parallel SAT Challenges

3.2.1 Resource allocation

As mentioned in this chapter, adding resources in SAT Solvers can increase their performance, due to searching in different sets of the search space, in case of a portfolio solver, or to a good split, in case of search-splitting solvers, which brings the solution to the beginning of a subspace. By contrast, adding less useful threads or a bad search space splitting could also lead to a bigger overhead, which could decrease the performance of a SAT solver.

The question of deciding which resources should be used in a parallel SAT Solver is a difficult challenge, given the different complexity of SAT instances.

3.2.2 Splitting

As mentioned previously in this chapter, there are two essential types of splitting algorithms in SAT solvers: search-space splitting algorithms, which divide the search-space into different subspaces, and problem splitting, in which the instance itself is decomposed in a way that none of the computing resources as access to the whole problem. Finding a decomposition in which the size of the subproblems is balanced can easily be done, but usually the hardness of these subproblems is heavily unbalanced. On the other hand, finding a decomposition that minimizes the number of shared variables is also a difficult problem and not necessarily the best approach.

Currently the state of the art for both types of decomposition is unsatisfactory. Further research is needed to find dynamic decomposition techniques that would lead to consistent results that

outperform currently known methods.

3.2.3 Knowledge Sharing

Most modern SAT solvers generate conflict clauses to prevent future conflicts and to prune the search-space. Parallel solvers usually share learned clauses to accelerate the search process. However, the number of generated clauses can be exponential and lead to a large overhead, in terms of memory (additional space required to clause allocation) and processing (Propagation and Backtracking has to be performed in a larger number of clauses). Sharing criteria have to be used to reduce this overhead.

A simple method to reduce the number of shared clauses is to limit the size of clause to a limit. This way, the number of shared clauses is heavily reduced and the cooperation is focused in more powerful clauses. Nevertheless, this static criteria could not be the most suitable in many occasions. For instance, if two threads explore completely different subsets of the search space, clause sharing would be useless. On the other hand, a deeper cooperation level would be adequate in two strategies that explore the same subset. Therefore, dynamic techniques should be used in order to improve the quality of the clause sharing methods.

3.3 Conclusion

In this chapter an overview of the current approaches to parallel SAT, as well as its challenges was presented.

Parallel SAT solving is a quite recent approach to SAT. Since 2008 there is a parallel track in the annual SAT competitions performed, and although parallel solvers outperform state-of-the-art sequential solvers, further research is needed to improve the effectiveness of these algorithms.

Given these methods and challenges, in the next chapters we will present our approaches to parallel SAT solving and describe the solvers developed in this thesis, PMCSAT and CLUSTERSAT.

4

pmcSAT

Contents

4.1 Description

Given the algorithms described in the previous sections and the predominance of portfolios over the other approaches, a portfolio solution with cooperative solving was developed: pmcSAT.

pmcSAT is a portfolio, multicore SAT solver that uses multiple threads to explore the search space independently. Each thread follows different paths due to the way each thread is configured. However, pmcSAT is not just a competitive SAT solver. The threads cooperate among themselves by sharing the learned clauses resulting from conflict analysis.

This portfolio solver was built on top of MiniSat 2.2.0 [11], a sequential state-of-the-art solver, developed in C++. The choice on this solver was based on its good performance, as well as in the fact that it is small, well-documented and easy to modify. The parallel approach to this solver was implemented using Pthreads. This solver was presented in the FLAIRS-26 conference [24], and won bronze medal in the Hard-Combinatorial Parallel Track of the SAT Competition of 2013 [2].

In the following sections we describe the adopted strategies in the implementation of pmcSAT.

4.1.1 Decision Heuristics

Decision heuristics are used on SAT solvers for selecting the next variable to be assigned, and the corresponding value, when no further propagation can be done. Although, some randomness is incorporated into most heuristics, we would like to keep a tight control over the search space explored by each thread. Therefore, the notion of variable priority was introduced. Variables with higher priority are assigned before variables with lower priority. The well proven VSIDS heuristic is used as the main decision heuristic but the variable selection is constrained by the priority assigned to each variable.

In order to ensure that each thread follows divergent search paths on the search space, we defined distinct priority assignment schemes, one for each thread of the pmcSAT solver. Table 4.1 describes the eight priority schemes that were used.

Note that, for most industrial SAT instances we can take advantage of the fact that the variables appear in the CNF file in a particular order, which is not random, but related to the problem structure. Therefore, threads 2-7 exploit that advantage by assigning priorities according to some variable properties.

In [1] the authors show that using a more agressive clause deletion strategy could lead to good results in a CDCL SAT solver, as a consequence of reducing the overhead on the propagation of learnt clauses. Therefore, thread #1 uses a more agressive deletion strategy, while all the other threads follow the original MiniSat deletion scheme.

4.1.2 Clause Sharing

It is well established that clauses learned during the search process, as a result of conflict analysis, are vital to speed up the search. In a parallel solver, the information learned from a

Thread #	Priority assignment scheme
0,1	All the variables have the same priority, therefore this thread mimics the original VSIDS heuristic.
2	The first half of the variables read from the file have higher priority than the second half.
3	The first half of the variables read from the file have lower priority than the second half.
4	The priority is sequentially decreased as the variables are read from the file.
5	The priority is increased according to the number of variable occurences in the file.
6	The priority is decreased according to the number of variable occurences in the file.
7	The priority is decreased according to the number of variables that have the same number of common variables.

Table 4.1: pmcSAT's priority assignment schemes for each thread

conflict in one particular thread can be very useful to other threads, in order to prevent the same conflict to take place.

Therefore, clause sharing between threads was implemented in pmcSAT. We limited the size of the clauses to be shared, to avoid the overhead of copying large clauses, which may contain very little relevant information. To reduce the communication overhead introduced by clause sharing, and its overall impact in performance, we designed data structures that eliminate the need for read and write locks. These structures are stored in shared memory, which is shared among all threads. We will consider that shared clauses are sent by a source thread and received by a target thread.

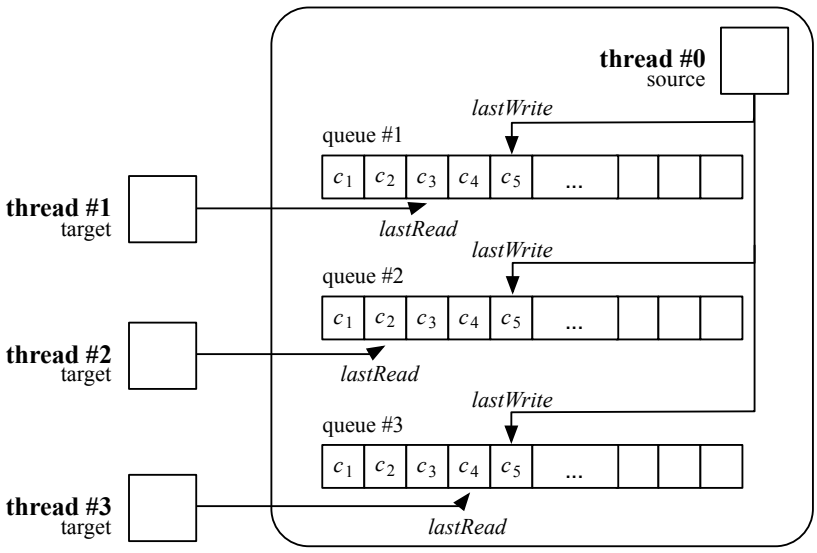

Figure 4.1: Clause sharing data structure of pmcSAT

As illustrated in Figure 4.1, each source thread owns a set of queues, one for each target thread, where the clauses to be shared are inserted. While this flexible structure enables sharing different clauses with different threads, we will restrict ourselves to sharing the same clauses with every

thread. Therefore, every thread is a source thread and their target threads are all the others. On each queue, the *lastWrite* pointer marks the last clause to be inserted. The *lastWrite* pointer is only written by the source thread, but can be read by each target thread. On the other hand, the *lastRead* pointer which marks the last clause received by the target thread, is only manipulated by each target thread. This data structure eliminates the need for a locking mechanism, since *lastRead* is only manipulated by one thread and even though *lastWrite* is read and written by different threads, the reading thread does not have to read its latest value. Clause sharing occurs after a conflict analysis.

4.2 Implementation

Algorithm 4.1 pmcSAT master process

```
1: Parse_CNF_File();
2: SolvedFlag = FALSE;
3: Launch_8_pmcSAT_Threads();
4: solution = Wait_For_First_Thread();
5: return SOLUTION;
```

Algorithm 4.2 pmcSAT thread

```
1: Configure_Thread(thread_ID);
2: while Performing_MiniSat() do
3:     Periodically_Check_Solved_Flag();
4:     if SolvedFlag == TRUE then
5:         Exit_Thread();
6:     end if
7:     if Conflict_Found() then
8:         Import_Conflict_Clauses();
9:         if ConflictClauseSize ≤ 8 then
10:            Export_Clause();
11:        end if
12:    end if
13:    if Solution_Found() then
14:        SolvedFlag = TRUE;
15:        Report_Solution_to_Master();
16:        Exit_Thread();
17:    end if
18: end while
```

Pseudocodes 4.1 and 4.2 describe the behavior of pmcSAT. The process starts with a master thread that parses the CNF file and launches 8 threads, already containing the information obtained from the instance. Each thread is configured according to the description given in the previous sections.

Followed by this configuration, the underlying solver of pmcSAT, MiniSat, starts. Whenever a thread bumps into a conflict, if the resulting conflict clause has a size ≤ 8 it is exported to the other threads. The thread also checks if any clauses have been exported by other threads and

imports them if so.

The first thread that finds a solution writes in a boolean flag that indicates if the problem has been solved, shared in memory by all threads. Every thread periodically checks that flag to confirm if any thread has already found a solution. If yes, the thread exits. The master process then waits for all threads to return and exits the process.

4.3 Results

4.3.1 Sequential solver/no sharing comparison

In this section we present preliminary results from applying pmcSAT to a slew of problems gathered from a recent SAT race [33]. We pay particular attention to those originated from circuit examples, given their practical relevance in an industrial setting, which are shown in Table 4.2. These include problems which are both known to be SAT or UNSAT. We run additional examples which are accounted for on the averages shown in the last line of Table 4.2. All experiments were conducted on a machine with a Dual Intel Xeon Quad Core processor at 2.33GHz, 24GB of RAM and running Fedora Linux, release 17.

Instance	Sol.	#Vars	#Clauses	MiniSAT (sec)	pmcSAT (sec)	Winning Thread	Speedup pmcSAT vs no sh	Speedup pmcSAT vs MiniSAT
cmu-bmc-barrel6	U	2306	8931	1.32	0.541	0,1,4	1.73	2.44
cmu-bmc-longmult13	U	6565	20438	26.27	4.753	7	4.58	5.53
cmu-bmc-longmult15	U	7807	24298	15.6	3.277	5,7	3.99	4.76
ibm-2002-11r1-k45	S	156626	633125	38.19	5.387	0,1,6	7.09	7.09
ibm-2002-18r-k90	S	175216	717086	102.3	18.593	1	5.50	5.50
ibm-2002-20r-k75	S	151202	619733	166.08	80.032	0,2	2.08	2.08
ibm-2002-22r-k60	U	208590	845248	716.53	210.817	6	3.40	3.40
ibm-2002-22r-k75	S	191166	793646	251.07	10.52	0	16.21	23.87
ibm-2002-22r-k80	S	203961	846921	159.03	44.15	1	1.30	3.60
ibm-2002-23r-k90	S	222291	922916	680.85	193.581	5	3.52	3.52
ibm-2002-24r3-k100	U	148043	545315	202.9	35.448	0	4.23	5.72
ibm-2002-30r-k85	S	181484	888663	850.73	128.671	7	6.61	6.61
ibm-2004-1_11-k80	S	262808	1023506	145.46	71.189	1	1.83	2.04
ibm-2004-23-k100	S	207606	847320	837.61	33.312	3	20.92	25.14
ibm-2004-23-k80	S	165606	672840	232.34	9.909	4	23.46	23.45
ibm-2004-29-k25	U	17494	74526	98.99	23.942	2	4.14	4.13
mizh-md5-47-3	S	65604	234719	265.53	34.715	1	7.65	7.65
mizh-md5-47-4	S	65604	234811	87	19.872	5	2.00	4.38
mizh-md5-47-5	S	65604	235061	563.58	28.42	4	4.56	19.83
mizh-md5-48-5	S	66892	240181	312.49	29.18	4	6.50	10.71
mizh-sha0-35-3	S	48689	173748	29.36	6.72	1,3	0.90	4.37
mizh-sha0-35-4	S	48689	173757	262.22	15.32	1	4.46	17.12
mizh-sha0-36-1	S	50073	179811	353.95	11.21	3,6	7.94	31.57
mizh-sha0-36-4	S	50073	179989	217.09	49.919	7	4.34	4.35
velev-engi-uns-1.0-4nd	U	7000	67553	10.83	3.246	0,1	3.15	3.34
velev-fvp-sat-3.0-b18	S	35853	1012240	27.05	2.278	6	11.89	11.87
velev-npe-1.0-9dlx-b71	S	889302	14582952	190.91	16.24	4	2.31	11.76
velev-vliw-sat-4.0-b4	S	520721	13348116	72.9	10.23	1,7	4.70	7.13
velev-vliw-sat-4.0-b8	S	521179	13378616	101.53	13.747	6	2.34	7.39
velev-vliw-uns-2.0-iq1	U	24604	261472	435.23	39.78	2,4	6.45	10.94
velev-vliw-uns-2.0-iq2	U	44095	542252	TO	321.24	2,4	5.12	NA
avg. table set (30 inst.)	–	155250	1752574	248.50	47.62	–	5.96	9.38
avg. full set (78 inst.)	–	168240	1119370	309.48	104.32	–	4.45	51.34

Table 4.2: Application of pmcSAT to a set of instances gathered from a SAT Race

In order, the columns of the table provide information about the problems chosen, namely the instance name, its solution and the problem size (number of variables and number of clauses). Following, the runtimes for MiniSat and pmcSAT using clause sharing are given. Also shown in

the table is the information of which thread(s) of pmcSAT first found a solution for each of the problems. When sharing of learned clauses is turned off, the comparison reduces to determining which of the strategies, described in Table 4.1, is more appropriate to a given problem or set of problems. In essence in this case all threads are instances of the basic MiniSat solver with the parameter configurations described. In this case, it turns out that the standard MiniSat performs quite well but many other strategies do equally well.

When sharing is turned on, the scenario changes considerably and picking chunks of variables from the top or bottom of the order seems to do quite well on many occasions, given the fact the most successful threads are #1 and #2.

Next we show speedup computations to attest the potential gains of our solver. First we compare pmcSAT with clause sharing versus non-clause sharing. When clause sharing is turned off we are really testing the appropriateness of the strategies in Table 4.1. When clause sharing is on, we are measuring the advantages of cooperation between threads. The advantages of clause sharing seem obvious: using information from other threads, which are exploring problem structure elsewhere in the search space, provides relevant information and speeds up problem solution. The speedup average when comparing pmcSAT with sharing vs. no sharing (4.45) confirm our statement. However the advantage of this approach is also offset by the cost of doing the sharing (both preparing clauses for sharing, as well as doing propagation in additional clauses). For this reason in certain problems little a small is obtained.

Analysing the data we can therefore conclude that pmcSAT is faster than the existing state of the art SAT solvers running on single core CPUs for real instance problems.

4.3.2 SAT Competition 2013 Results

In this section the results of pmcSAT from the 2013 SAT Competition [2] are presented. The solver was submitted to two different parallel tracks: Application and Hard-Combinatorial.

All experiments were conducted on the bwGRiD cluster of the State of Baden-Württemberg, Germany. The cluster nodes have the following specification: 2x Quad-Core Intel Xeon E5440 processors, at 2.83 GHz, with 16GB of RAM per node and running Scientific Linux, kernel 2.6.18. The Computational resources in the parallel Tracks were 8 cores (of a cluster node), 15GB RAM, 5000 seconds wall-clock time, 100 GB /tmp disk space.

In each track the solvers were tested on 300 instances. The following sections describe the results in detail. The classification criteria is the number of solved instances, and the total wall clock time used to computed the solutions as a tiebreaker. The first, second and third places are awarded with a gold, silver and bronze medal, respectively.

#	Solver version	#solved	% of all	% of VBS	#runs used by VBS	Average Wall Time
-	Virtual Best Solver (VBS)	290	96.7	100.00	290	282.10
1	Plingeling aqw	271	90.3	93.45	60	1026.79
2	Treengeling aqw	260	86.7	89.66	36	1261.10
3	PeneLoPe 2013	247	82.3	85.17	26	1628.56
4	Glucans strict	241	80.3	83.10	60	1489.27
5	pcasso port	227	75.7	78.28	9	1769.86
6	pcasso 1	227	75.7	78.28	7	1781.16
7	SatX10-GLCI bugfix	226	75.3	77.93	15	1768.70
8	strangenight satcomp11-mt	225	75.0	77.59	19	1937.02
9	**pmcSAT 1.0**	**213**	**71.0**	**73.45**	**18**	**2158.41**
10	GlucoRed r531	210	70.0	72.41	28	2019.33
11	GlucoRed-Multi r531	205	68.3	70.69	10	2159.68
12	SAT4J Parallel SC2013	111	37.0	38.28	2	3750.11

Table 4.3: SC2013: Parallel Application Track Results

Application Track

This track refers to problem encodings (both SAT and UNSAT) from real-world applications, such as hardware and software verification, bio-informatics, planning, scheduling, etc. Table 4.3 lists the parallel solvers that entered this track and its results. On the first column is the solver final ranking. Next, the number of solved instances (out of 300) by each solver are given, followed by the total % of instances in which the solver could find a solution.

The **Virtual Best Solver (VBS)** is a theoretical construction which returns the best answer provided by one of the submitted solver. The next two columns compare the solvers with the VBS, showing the total % of instances solved by the VBS in which the solver could find a solution, as well as the number of instances where the solver had the best result, being therefore picked by the VBS.

From the table, we can observe that Plingeling aqw and Treengeling aqw, both from the same author, Armin Biere, had been awarded with a gold and silver medal, with 271 and 260 instances solved. PeneLoPe 2013, with 247 instances solved, has been awarded with a bronze medal. pmcSAT was ranked 9^{th} with 213 instances, which is not a particularly remarkable classification, given the fact that there are 12 competitors.

Analysing the average wall time, as it would be expected, these times follow an inverse ratio with the number of solved instances. As the number of solved instances increases, the average wall time is expected to decrease.

Although Glucans strict was classified 4^{th}, it is the solver that has been more often picked by the VBS, along with Plingeling, with 60 instances each, while the third solver that has been more picked, Treengeling, has been the best solver in 36 instances, a slightly lower value, when comparing to the first pair. pcasso solver, both port and 1 versions, in spite of being classified 5^{th} and 6^{th} are among the solvers that were less often picked by the VBS (9 and 7 instances picked).

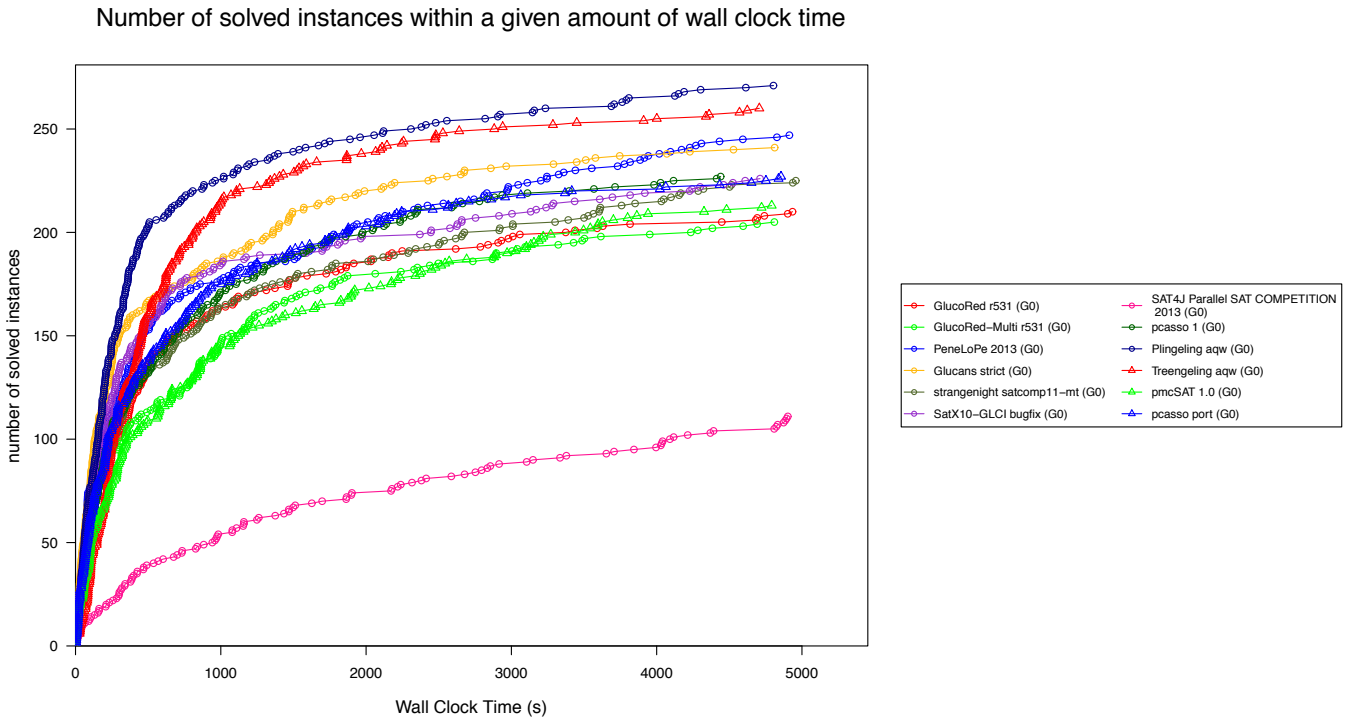

Figure 4.2: Number of solver instances solved by each solver within a given time

Figure 4.2 shows the number of instances solved by each solver within a given time. From this point of view we can easily conclude that all solvers find a solution for most instances in less than 1000 s. From that point on the solvers try to solve harder instances that could take much more time, therefore, the solving ratio decreases significantly.

Hard-combinatorial Track

This track refers to instances that are often designed to give a hard time to SAT solvers (including, e.g., instances arising from difficult puzzle games). Table 4.4 describes the results this track.

#	Solver version	#solved	% of all	% of VBS	#runs used by VBS	Average Wall Time
-	Virtual Best Solver (VBS)	276	92.0	100.00	276	303.82
1	Treengeling aqw	253	84.3	91.67	64	1149.49
2	Plingeling aqw	242	80.7	87.68	25	1412.73
3	pcasso (disqualified) port	220	73.3	79.71	22	1917.39
4	**pmcSAT 1.0**	**219**	**73.0**	**79.35**	**48**	**1895.64**
5	pcasso (disqualified) 1	219	73.0	79.35	30	1916.03
6	Glucans strict	206	68.7	74.64	29	2038.57
7	GlucoRed r531	204	68.0	73.91	27	2055.15
8	GlucoRed-Multi r531	197	65.7	71.38	6	2173.52
9	strangenight satcomp11-mt	189	63.0	68.48	23	2214.07
10	SAT4J Parallel SC2013	117	39.0	42.39	2	3388.39

Table 4.4: SC2013: Parallel Hard-Combinatorial Track Results

The structure of Table 4.4 is the same as table 4.3. As we can see, TREENGELING AQW and PLINGELING AQW again outperform the other parallel SAT Solvers. Although PCASSO PORT solves one more instance than PMCSAT, it has been disqualified due to producing a wrong answer to an instance. Therefore, PMCSAT was placed third in this track, winning a bronze medal.

PMCSAT has a similar performance when comparing with the application track (219 instances solved against 213), while all the other solvers that also compete in the Application Track show a slight decrease of performance. It also is the best solver for 48 instances, only being overcame by TREENGELING, the best solver in 64 instances. We can thereby conclude that PMCSAT is suited for difficult instances.

Figure 4.3 shows the number of instances solved by each solver within a given time. Besides PLINGELING and TREENGELING, all the other solvers performances have similar performances until about 1000 s, with the exception of SAT4J, which solves much less instances than any other solver. With exception to TREENGELING and PLINGELING, PMCSAT reveals a better performance than the other solvers through almost the all execution time, since about 1000 s. As mentioned previously, although PCASSO PORT solves one more instance than PMCSAT, it has been disqualified due to producing a wrong answer to two instances.

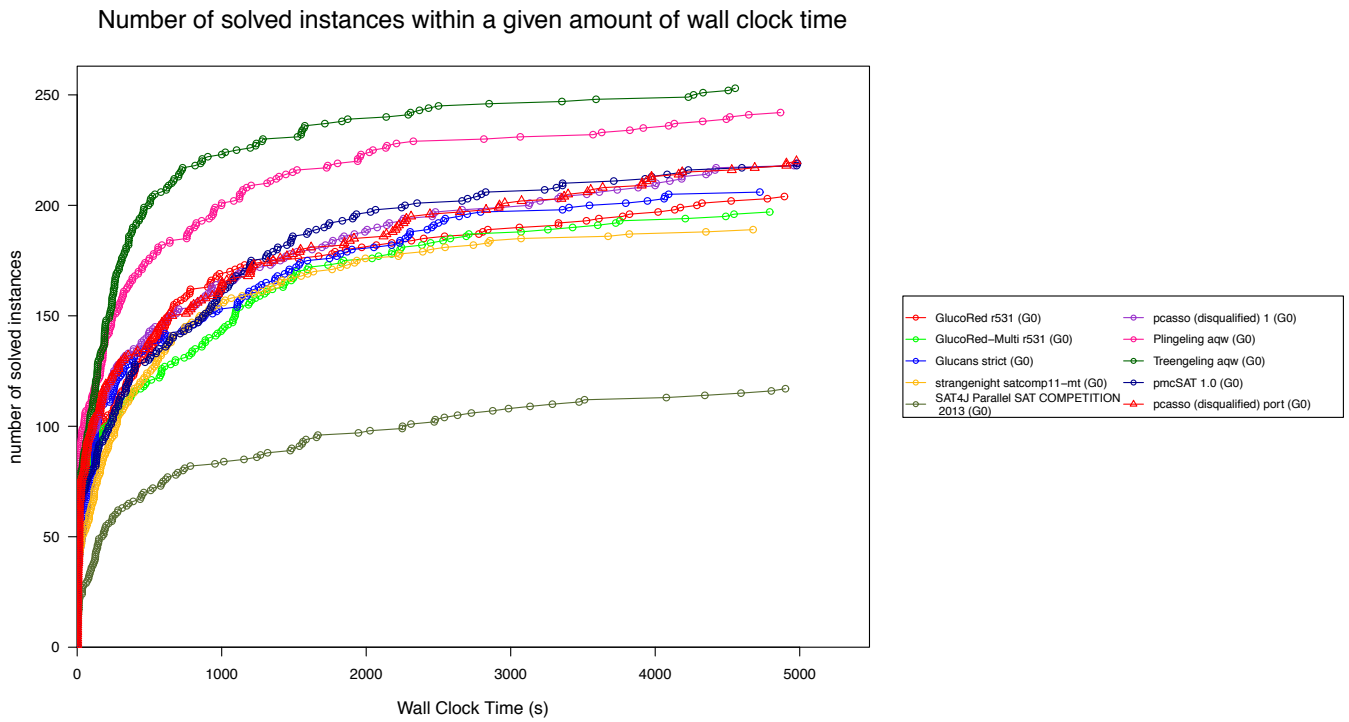

Figure 4.3: Hard-Combinatorial: Number of solver instances solved by each solver within a given time

5

clusterSAT

Contents

5.1 Description

After developing a portfolio SAT Solver, research was done in the problem splitting approach, in which the state of the art is unsatisfactory. As mentioned in chapter 3, this approach is based on splitting the original formula into several subformulas. These could be solved in parallel, although there is the need to have synchronism due to the shared variables between subformulas.

A SAT solver based on this techniques was developed: clusterSAT. This solver was also built on top of MiniSat 2.2.0 with Pthreads for parallel implementation. The algorithm is divided in two distinct fases: first the **clustering** phase, in which the problem is split, and then the **solving** phase, in which the different clusters are solved and synchronised.

5.1.1 Clustering

Finding a decomposition where the size of the subproblems is balanced is easy, but balancing the hardness of these subproblems, as well as minimising the number of shared variables between clusters is also a difficult challenge.

This decomposition was done using the Metis [18] and hMetis [17] libraries.

Metis Decomposition

Metis is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices. The algorithms implemented in Metis are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes.

Metis accepts as input an unidirectional graph. It was considered that each clause is a node from the graph, in which the edges correspond to common variables between clauses. A weight can be assigned to each edge, being related to the number of shared variables between clauses. The program output is an assignment of each node (clause) to a cluster. The number of desired clusters is defined by the user.

hMetis Decomposition

hMetis is a set of programs for partitioning hypergraphs. A hypergraph is a generalization of a graph in which an edge can connect any number of vertices. Therefore, it was considered an approach where the nodes of the hypergraph would be the clauses of the problem, and the hyper-edges would be the variables, containing all the clauses where it appears. The program output is equal to the output of the Metis approach: an assignment of each clause to a cluster.

Figure 5.1 illustrates a simple example of a conversion from a CNF formula to a Metis and hMetis input. Edges in graph a) connect clauses that share the same variable, while in graph b) edges connect all the clauses in which a variable appears. For instance, x_1 is shared between

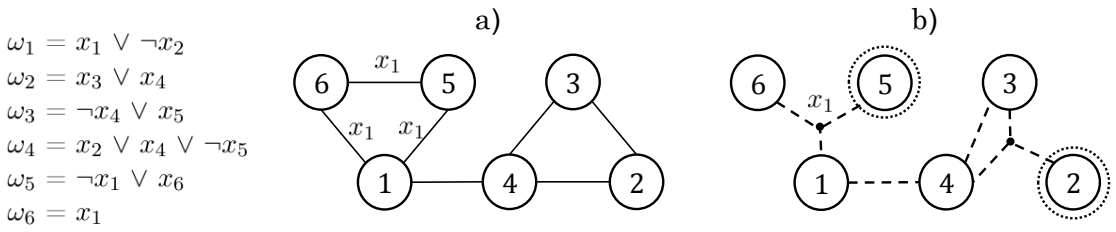

Figure 5.1: CNF formula conversion to a) METIS, b) hMETIS

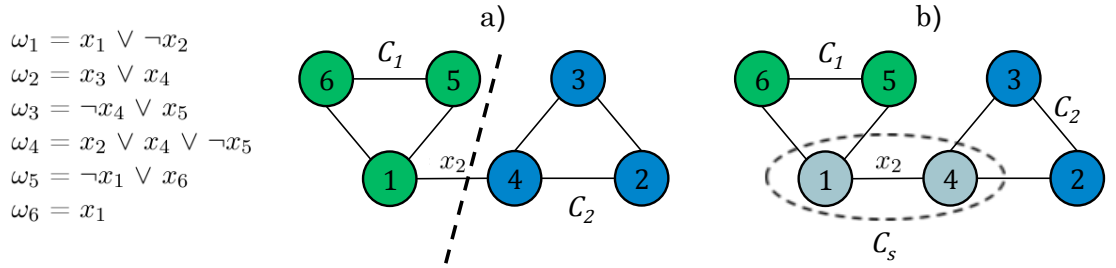

Figure 5.2: Partition obtained from the clustering algorithms

clauses 1, 5 and 6. In graph *a*) all those nodes are connected among themselves, being 3 edges needed, while in graph *b*) a single hyper-edge connects the three nodes.

As we can we observe in figure 5.2, clauses 1, 5 and 6 are strongly connected, as well as clauses 2, 3 and 4. Two types of outcoming clusters were considered:

- **Disjoint clusters** - In this kind of partition each cluster would have the clauses assigned by the clustering program output (e.g., in figure 5.2 *a*, nodes 1, 5 and 6 belong to C_1 and nodes 2, 3 and 4 belong to C_2). The only variable **shared** between these clusters is x_2, which would need special attention during the solving process. All the other variables are considered to be **local**, since they only appear in a single cluster.

- **Centered clusters** - Similar to the disjoint clusters partition, but with an additional cluster, C_s, that is composed by the clauses which contain variables that belong to more than one cluster. This way, in a partition with N clusters, $C_1, C_2, ..., C_n$ don't have any type of connection, only sharing variables with C_s (e.g., in figure 5.2 *b*, nodes 1 and 4 are removed from C_1 and C_2, respectively and would compose a new cluster C_s). This type of partition is similar to the approach in JACK-SAT [32].

After the instance decomposition, the solving process is started. In the following section the different approaches, which use both types of decomposition, are described in detail.

5.1.2 Solving

To tackle the problem, different master-slave approaches were considered for the two types of partitions. In all of these, the roles of the master/slaves were the following:

- **Master** - Thread that controls the execution of the whole solving process. Its role is to assure a consistent state, regarding the assignments of the shared variables.
- **Slaves** - Each slave is assigned to a cluster. Their role is to solve the cluster and report the search-state to the master thread.

The different algorithms implemented were the following:

Master-First, Disjoint Clusters

In order to avoid conflicting assignments of the shared variables, the master has access to the whole problem and performs these assignments *a priori*. This is done by giving higher priority to shared variables using the same scheme as in PMCSAT. Once these variables are assigned, the slaves start to solve their respective clusters. Their work will result in one of the following outcomes:

- The slave finds a solution to its cluster and reports **SAT** to the master.
- The slave concludes that the problem is **UNSAT** and reports it to the master.
- The slave finds a conflict within the shared variables' level and reports **UNKNOWN** to the master, and also the respective **conflict clause**.

After all the slaves finish their work the master resumes its work, performing an analysis of the work reported. This analysis will result in one of the following:

- If all the slaves return SAT then the problem is **SATISFIABLE**.
- If one or more slaves return UNSAT then the problem is **UNSATISFIABLE**.
- Otherwise, the master imports the conflict clauses exported by the slaves, redoes the assignment of the shared variables and the slaves restart with the new set of shared assignments.

Master-First, Centered Clusters

In this approach, no thread has access to the whole problem. For instance, in a N cluster partition, each slave will solve its respective cluster, and the master process will be responsible for the residual cluster, C_s.

This process is very similar to the previous one, with the exception of the master's solving scheme. Instead of having access to the whole problem and assigning only the shared variables between clusters, the master solves its own cluster and exports the assignments to the slaves.

The behavior of the algorithm in the rest of the process is equal to the previously described.

Figure 5.3 resumes the behavior of the algorithm in both types of partitions.

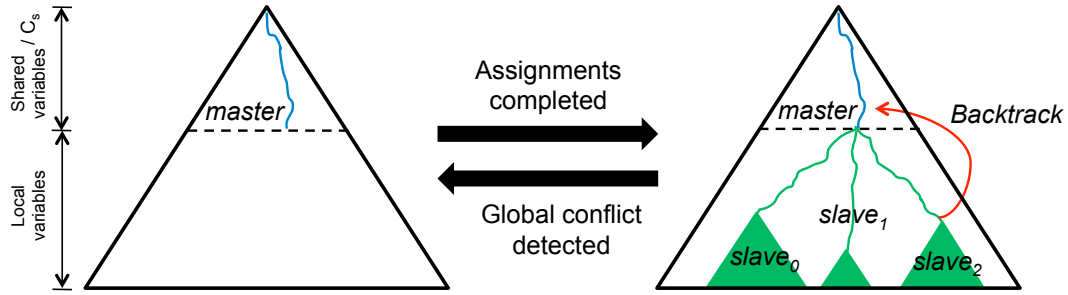

Figure 5.3: Master-first solving process description

Slaves-First, Disjoint Clusters

This approach is slightly different than the ones described previously. Instead of being the master to start the solving process, the slaves solve their clusters in parallel.

The role of the master in this approach is to analyse the slaves' guiding paths and undoing opposite assignments, regarding the shared variables. A set of patterns is used to weight the quality of each assignment:

- The **polarity** of the variable assignments. Let A_P be the number of *true* assignments of the variable and A_N the number of *false* assignments. The variable is more likely to be assigned *true* if $A_P \geq A_N$, and assigned *false* otherwise.
- The associated **decision level** of the variable. The assignment is more likely to be considered if its corresponding decision level is lower, which means that the variable is in a higher level of the decision tree.
- The number of **implications** generated by the variable assignment. The assignment is more likely to be considered if it generates more implications.

The outcoming set of assignments is then exported to the slaves and the process restarts, until there are no different assignments among the shared variables. The problem is then considered **SATISFIABLE**. The problem is considered **UNSATISFIABLE** when a slave detects that its cluster is UNSAT, or when the master process detects conflicting unit assignments on the top of the decision tree.

Slaves-First, Centered Clusters

Again, this approach is very similar to the previous one with exception to the master role. Since with this kind of partition the only shared variables are between the residual cluster C_s and other clusters, the slaves do not share any variables. Therefore, when the master process imports the slaves' guiding paths there would be no conflicting assignments.

However, the master has to solve the residual cluster. If a conflict occurs on the slaves' assignments level, the respective conflict clauses are exported and all the process restarts. If the master

process finds a solution for its cluster then the problem is **SATISFIABLE**. Again, the problem is reported **UNSATISFIABLE** when one of the clusters, including the master's cluster, returns UNSAT in anytime of the search-process.

Figure 5.4 describes graphically this approach.

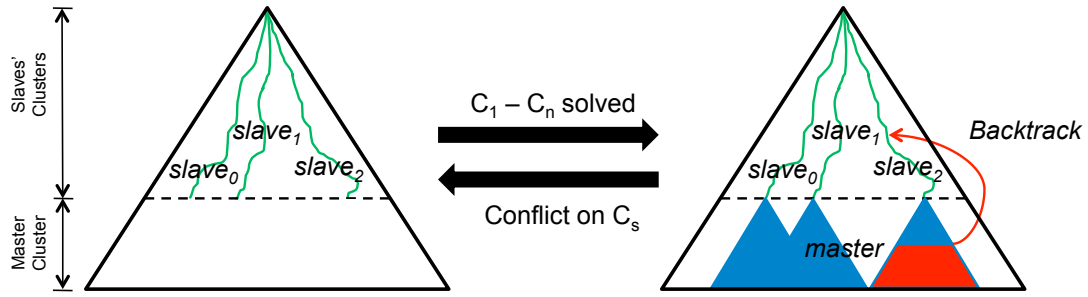

Figure 5.4: Slaves-first, Centered Clusters solving process description

5.2 Results

In this section the experimental results from applying clusterSAT to a set of CNF instances are presented. Given the novelty of the approach and the difficulty of partitioning large problems, we focused on solving problems of medium difficulty, in which a good partition could be done. The approach with the best performance was by far the **Master First, Centered Clusters**, which is thus the approach considered for the results presented. All experiments were conducted on a machine with a Dual Intel Xeon Quad Core processor at 2.33GHz, 24GB of RAM and running Fedora Linux, release 17.

Instance	Sol.	#Vars	#Clauses	% Shared Vars			% Clauses w/shared vars			% Shared Vars/Clause		
				2 cl.	4 cl.	8 cl.	2 cl.	4 cl.	8 cl.	2 cl.	4 cl.	8 cl.
cmu-bmc-barrel6.cnf	U	2306	8931	12.14%	25.59%	40.16%	20.11%	49.08%	70.34%	13.37%	25.59%	46.97%
cmu-bmc-longmult13.cnf	U	6565	20438	2.03%	4.71%	9.58%	4.30%	9.15%	24.14%	2.47%	4.91%	13.27%
cmu-bmc-longmult15.cnf	U	7807	24298	1.01%	4.05%	5.97%	1.82%	8.37%	14.61%	0.86%	5.05%	7.43%
bmc-ibm-1.cnf	S	9685	55180	1.17%	2.89%	7.06%	3.14%	10.63%	21.36%	1.36%	5.49%	11.65%
bmc-ibm-2.cnf	S	2810	11185	2.28%	1.53%	5.05%	8.24%	6.76%	27.61%	5.91%	3.11%	16.94%
bmc-ibm-3.cnf	S	14930	71533	1.25%	2.10%	6.79%	2.47%	4.24%	15.37%	1.22%	2.04%	7.91%
bmc-ibm-4.cnf	S	28161	134213	0.69%	1.86%	4.18%	2.74%	7.13%	16.16%	1.21%	3.35%	8.30%
bmc-ibm-5.cnf	S	9396	40534	0.73%	2.47%	3.82%	3.34%	11.27%	15.82%	1.29%	4.91%	6.79%
bmc-ibm-6.cnf	S	51639	359916	0.52%	1.39%	2.58%	1.43%	4.26%	8.12%	0.68%	1.94%	3.69%
bmc-ibm-7.cnf	S	8710	38542	2.19%	4.95%	7.03%	9.04%	23.90%	32.58%	4.42%	14.60%	18.73%
bmc-galileo-8.cnf	S	58074	291715	0.39%	1.10%	2.73%	0.69%	2.62%	7.59%	0.31%	1.06%	2.90%
bmc-galileo-9.cnf	S	63624	323881	0.38%	1.03%	2.29%	1.18%	3.23%	6.56%	0.48%	1.16%	2.47%
bmc-ibm-10.cnf	S	59506	314563	0.60%	2.20%	3.72%	1.94%	7.47%	11.84%	0.81%	3.88%	5.55%
bmc-ibm-11.cnf	S	32109	148281	0.29%	1.67%	2.76%	0.74%	4.70%	8.18%	0.32%	2.62%	3.82%
bmc-ibm-12.cnf	S	39598	193434	0.32%	0.95%	2.19%	0.86%	2.31%	5.05%	0.36%	0.96%	2.11%
bmc-ibm-13.cnf	S	13215	65025	1.45%	2.63%	10.36%	3.36%	6.30%	25.85%	1.59%	3,12%	16.02%
een-tip-sat-nusmv-t5.B.cnf	S	61933	166069	0.40%	1.10%	2.83%	0.68%	3.69%	9.69%	1.41%	1.73%	4.64%
een-tip-sat-texas-tp-5e.cnf	S	17985	47464	0.33%	1.24%	2.67%	0.86%	3.19%	7.53%	0.41%	1.61%	3.66%
een-tip-sat-vis-eisen.cnf	S	18607	52440	0.21%	0.61%	2.76%	0.35%	1.83%	7.54%	0.16%	0.89%	3.96%
avg. table set(19 inst.)	-	26666	124613	1.49%	3.37%	6.55%	3.54%	8.95%	17.68%	2.03%	4.72%	9.83%

Table 5.1: Application of Metis to a set of CNF instances

In table 5.1, the clustering results from applying Metis are given. Due to the high run-times of hMetis, only the Metis tool was used to obtain a partition. First, the name of the instance

is presented, as well as its outcome and the respective number of variables and clauses. Next the decomposition results, for 2, 4 and 8 clusters are described, namely the ratio of shared variables, clauses with shared variables, and number of shared variables per clause.

As it would be expected, the number of shared variables increases reasonably with the number of clusters. Consequently, the number of clauses with shared variables raises, as well as the the shared variables per clause. Comparing these values, we can also conclude that the shared variables grow in approximately the same ratio as the shared variables per clause, while the ratio of clauses with shared variables grows faster, since each clause has several literals.

Instance	Sol.	#Vars	#Clauses	Clustering (sec)			MiniSat	Solving (sec)			Best Speedup	
				2 cl.	4 cl.	8 cl.	(sec)	2 cl.	4 cl.	8 cl.	Total time	Solving
cmu-bmc-barrel6.cnf	U	2306	8931	0.04	0.07	0.20	1.32	2.07	1.11	0.70	1.47	1.90
cmu-bmc-longmult13.cnf	U	6565	20438	0.05	0.05	0.07	26.27	42.71	34.34	23.75	1.10	1.11
cmu-bmc-longmult15.cnf	U	7807	24298	0.06	0.06	0.08	15.60	63.14	47.21	22.09	0.70	0.71
bmc-ibm-1.cnf	S	9685	55180	0.11	0.12	0.17	0.05	0.15	0.27	0.22	0.18	0.32
bmc-ibm-2.cnf	S	2810	11185	0.01	0.01	0.01	0.01	0.00	0.00	0.00	0.78	2.33
bmc-ibm-3.cnf	S	14930	71533	0.15	0.17	0.22	0.12	0.15	0.17	0.14	0.38	0.86
bmc-ibm-4.cnf	S	28161	134213	0.16	0.16	0.28	0.12	0.28	0.35	0.20	0.28	0.60
bmc-ibm-5.cnf	S	9396	40534	0.07	0.07	0.10	0.02	0.04	0.03	0.01	0.24	2.09
bmc-ibm-6.cnf	S	51639	359916	0.98	1.46	1.16	0.17	0.19	0.14	0.11	0.14	1.56
bmc-ibm-7.cnf	S	8710	38542	0.03	0.04	0.06	0.01	0.01	0.01	0.00	0.41	3.50
bmc-galileo-8.cnf	S	58074	291715	2.00	2.31	3.10	0.24	0.35	0.28	0.15	0.10	1.57
bmc-galileo-9.cnf	S	63624	323881	2.32	2.71	2.54	0.17	0.20	0.16	0.10	0.07	1.71
bmc-ibm-10.cnf	S	59506	314563	1.01	1.12	1.51	0.57	1.82	1.15	0.75	0.25	0.75
bmc-ibm-11.cnf	S	32109	148281	0.32	0.27	4.33	0.26	7.90	4.54	1.78	0.05	0.15
bmc-ibm-12.cnf	S	39598	193434	0.86	0.91	0.73	2.33	8.27	6.40	2.55	0.71	0.91
bmc-ibm-13.cnf	S	13215	65025	0.18	0.15	0.21	2.48	13.85	3.21	0.17	6.53	14.91
een-tip-sat-nusmv-t5.B.cnf	S	61933	166069	0.37	0.36	0.55	3.63	4.17	3.38	3.02	1.02	1.20
een-tip-sat-texas-tp-5e.cnf	S	17985	47464	0.07	0.06	0.07	0.18	0.26	0.18	0.12	0.95	1.56
een-tip-sat-vis-eisen.cnf	S	18607	52440	0.07	0.08	0.09	0.34	3.73	1.28	0.32	0.83	1.08
avg. table set(19 inst.)	-	26666	124613	0.47	0.53	0.82	2.84	7.86	5.48	2.96	0.85	2.04

Table 5.2: Application of ClusterSAT (Master-first, Centered Clusters) to a set of CNF instances

Table 5.2 describes the results from applying clusterSAT to the given set of instances. Again the description of the instance is given, followed by the time used by Metis to perform a partition of the problem (in 2, 4 and 8 clusters), followed by the running time of the underlying algorithm, MiniSat, and the solving time used by clusterSAT to solve the problem, using 2, 4 and 8 clusters. Finally, the speedup obtained comparing clusterSAT to MiniSat, considering all the execution time (Clustering+Solving), and just the Solving time.

As we raise the number of clusters, the clustering time used by Metis also increases. The solving time, however, decreases remarkably, which suggests that increasing the number of clusters would benefit the performance of clusterSAT. Analysing the obtained speedups, considering just the solving time, the algorithm obtains some speedup in several instances. Here we used the time for the best clustering, 2, 4 or 8. Usually it is 8. However, if we consider all the processing time, most of these speedups are lost.

We can therefore conclude that the results of applying clusterSAT don't show any major improvements yet. MiniSat, the base solver for this strategies, outperforms any of the different strategies implemented. The identified problems that resulted in this decrease of performance were the following:

- A big **unbalanced workload** was verified through the execution of clusterSAT. When the master thread is performing variable assignments, all the slaves are idle, waiting for a work order. Also, the master only restarts its work when he receives a response from all the slaves, which means that slaves that finish solving their clusters earlier will be idle until all are finished. This was an implementation choice that can be improved at a later time.

- **Repeated iterations** of the algorithm with a lot of **redundant work** performed. This issue happens in all the strategies implemented in the following way.

  - **Master-first, Disjoint Clusters** - The master process assigns all the shared variables and the slaves identify a single conflict on the assignments. The master has to import the conflict clause and reassign all the variables.
  - **Master-first, Centered Clusters** - The master solves its cluster and the slaves detect a conflict on the assignments. The master process has to import the conflict clause and solve the whole cluster again.
  - **Slaves-first** - The slaves solve their respective clusters and the master detects a conflict on the shared variables assignment. The slaves may have to reassign all their variables. This happens in both Disjoint Clusters and Centered Clusters approaches.

  In all these approaches, much of the assigns performed after a detection of a conflict are the same, which is the main reason for the low performance of this solver.

- **Forced modification** of the decision heuristic. In the **Master-First, Disjoint Clusters** approach, higher priority has to be assigned to the shared variables between clusters. The outcoming of this method is generally worse than the original VSIDS, resulting often in a decrease of overall performance.

- In the **Slaves-first** approaches, a very common issue results in the repetition of the same assignments in successive iteration. This is due to an insufficient conflict analysis from the master. An example of this issue is detailed below:

  $$C_1 = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_5), \;\; \rho(C_1) = \{x_1 = 0\}$$

  $$C_2 = (x_3 \vee x_4) \wedge (\neg x_4 \vee x_5), \;\; \rho(C_2) = \{x_3 = 0\}$$

  In this case, the $\rho(C_1)$ and $\rho(C_2)$ lead to conflicting assignments, $x_5 = 0$ and $x_5 = 1$. A standard analysis from the master process leads to the following conflict clause: $(x_1 \vee x_3)$. This clause, from the point of view of a slave is useless, since it contains variables from other clusters that would never be assigned. Therefore this clause would never to drive an implication and avoid a conflict.

From the issues described, we can conclude that this approach is still in its infancy. Further research is needed to improve the algorithm, both in the clustering and solving sections.

6

Conclusions

Contents

Boolean Satisfiability (SAT) is fundamental procedure in solving a multitude of problems and has applications in a variety of fields of Computer Science and Engineering. However, in the past decades several new technological novelties have enabled unprecedented advances in sequential SAT solving. These include non-chronological backtracking, restarts, improvements in decision heuristics, etc. However, in recent years this trend of continuous new developments has slowed down. Unfortunately this is at odds with growing appetite for faster solvers able to solve larger, harder problems. New approaches are thus required to enable this continued growth in solver capacity and performance.

The widespread availability of parallel parallel environments, such as multi-core with shared memory provides an opportunity for boosting the effectiveness of SAT solution. In the last years the number of cores and processors has been continuously increasing.

The SAT community is becoming more interested in parallel approaches. Since 2008, there has been a parallel track in the annual competition for parallel SAT solvers, which is designed for shared memory architectures. Currently, portfolio approaches are dominating this track. Even though the increase in the performance is not as significant as expected, modern parallel SAT solvers can now outperform state-of-the-art sequential solvers.

6.1 Contributions

In this dissertation two different parallel multi-core SAT solvers where developed to speed-up the search for a problem solution. These solvers endorse two different approaches to parallel solutions of SAT problems.

pmcSAT

pmcSAT is a portfolio SAT solver, which launches multiple instances of the same solver, with different parameter configurations. These tasks cooperate to a certain degree by sharing relevant information when searching for a solution. Different decision heuristics are set, combined with a lockless clause sharing mechanism. Although it is not a novel contribution, this algorithm uses a priority-based decision heuristic schema, a innovating concept upon portfolio SAT solvers.

Experimental results show that pmcSAT is a high quality parallel SAT solver, since it outperforms current state of the art sequential solvers. It has also been awarded with a bronze medal in one of parallel tracks of SAT competition 2013 [2] and has been presented in Florida, USA, in the FLAIRS-26 conference [24].

clusterSAT

clusterSAT uses graph partitioning libraries to split the original problem into smaller subproblems and launches multiple threads to solve these subproblems, that cooperate to find a valid solution.

Experimental results from applying clusterSAT to instances from real-world applications do not show major improvement over state of the art sequential solvers, which is somewhat upsetting. However we point out that this is a novel approach still somewhat in its infancy. The clustering and solving techniques are still inefficient but further research is needed to improve the performance of this algorithm and with the description of our different approaches, we hope to enhance the development of new techniques that may possibly outperform current state of the art solvers.

6.2 Future work

Parallel SAT solving is still a recent field of research on propositional satisfiability. Over the past years, computational resources have evolved in a way that higher computational power comes from a higher number of processor cores and therefore SAT solving has had a major focus in parallel techniques.

pmcSAT, in spite of this showing already remarkable results, could still take some modifications in order to improve its performance:

- As mentioned in this thesis, dynamic clause sharing could improve the performance of a parallel SAT solver. So, in threads which use similar search approaches, the clause sharing schema between threads should be deeper, otherwise it should be less deep.
- Additional parameters could be tuned (e.g., restarts, learning schema, etc.)
- Use different sequential solvers as underlying SAT solver (e.g., glucose [1]).

clusterSAT is still in an infant state, so several modifications would be needed in order to improve its performance. The most important aspects are described with detail in section 5.2:

- A big unbalanced workload was verified through the execution of clusterSAT.
- Repeated iterations of the algorithm with a lot of redundant work performed.
- Forced modification of the decision heuristics.
- Repeated iterations of the algorithm due to insufficient conflict analysis.

Concluding, parallel SAT solving has shown major improvements over the past years. Current state of the art parallel solvers outperform any sequential approach. With multicore architectures becoming predominant, it is important to continue exploring parallel techniques and frameworks that take advantage of this reality in order for SAT solvers to continue to evolve.

Bibliography

[1] Audemard, G. and Simon, L. (2009). Predicting learnt clauses quality in modern sat solver. In Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09), pages 399–404.

[2] Balint, A., Belov, A., Heule, M., Järvisalo, M., and et al (2013). SAT Competition 2013. http://satcompetition.org/2013/.

[3] Biere, A. (2008). Picosat essentials. Journal on Satisfiability, Boolean Modeling and Computation (JSAT).

[4] Biere, A. (2010). Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical Report 10/1, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Austria.

[5] Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. (1999). Symbolic Model Checking without BDDs. In Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS.

[6] Buro, M. and Büning, H. K. (1993). Report on a SAT competition. Bulletin of the European Association for Theoretical Computer Science, 49:143–151.

[7] Chu, G., Stuckey, P. J., and Harwood, A. (2008). Pminisat: A parallelization of minisat 2.0. Technical report, NICTA Victoria Laboratory.

[8] Cook, S. A. (1971). The complexity of theorem-proving procedures. In Proceedings of the Third IEEE Symposium on the Foundations of Computer Science, pages 151–158.

[9] Davis, M., Logemann, G., and Loveland, D. (1962). A Machine Program for Theorem-Proving. Communications of the ACM, 5(7):394–397.

[10] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. J. ACM, 7(3):201–215.

[11] Een, N. and Sorensson, N. (2004). An Extensible SAT-solver. Theory and Applications of Satisfiability Testing, 2919:502–518.

[12] Freeman, J. W. (1995). Improvements to Propositional Satisfiability Search Algorithms. PhD thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia.

[13] Goldberg, E. I., Prasad, M. R., and Brayton, R. K. (2001). Using sat for combinational equivalence checking. In DATE, pages 114–121.

[14] Gomes, C. P., Selman, B., and Kautz, H. (1998). Boosting Combinatorial Search Through Randomization. In Proceedings of AAAI, pages 431–437, Madison, Wisconsin, USA.

[15] Hamadi, Y., Jabbour, S., and Sais, L. (2009). ManySAT: A Parallel SAT Solver. JSAT, 6.

[16] Jeroslow, R. G. and Wang, J. (1990). Solving propositional satisfiability problems. Annals of Mathematics and Artificial Intelligence, 1:167–187.

[17] Karypis, G. and Kumar, V. (2011a). hMETIS, A Hypergraph Partitioning Package Version 1.5.3. http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download.

[18] Karypis, G. and Kumar, V. (2011b). METIS, A software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 5.0. http://glaros.dtc.umn.edu/gkhome/metis/metis/download.

[19] Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., and Selman, B. (2002). Dynamic Restart Policies. In Proceedings of AAAI, pages 674–681, Edmonton, Alberta, Canada.

[20] Kim, J., Silva, J. P. M., Savoj, H., and Sakallah, K. A. (1997). Rid-grasp: Redundancy identification and removal using grasp. In In IEEE/ACM International Workshop on Logic Synthesis.

[21] Kottler, S. and Kaufmann, M. (2011). SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In Pragmatics of SAT.

[22] Larrabee, T. (1992). Test pattern generation using boolean satisfiability. IEEE Transactions on Computer-Aided Design, 11:4–15.

[23] Lewis, M., Schubert, T., and Becker, B. (2007). Multithreaded sat solving. In Proceedings of the 2007 Asia and South Pacific Design Automation Conference, ASP-DAC '07, pages 926–931, Washington, DC, USA. IEEE Computer Society.

[24] Marques, R. S., e Silva, L. G., Flores, P., and Silveira, L. M. (2012). Improving sat solver efficiency using a cooperative multicore approach. International FLAIRS Conference, May 22 - 24, 2013.

[25] Martins, R., Manquinho, V., and Lynce, I. (2012). An overview of parallel sat solving. Constraints, 17(3):304–347.

[26] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In Proceedings of DAC, pages 530–535, Las Vegas, Nevada, USA.

[27] Roussel, O. (2011). Description of ppfolio.

[28] Schubert, T., Lewis, M. D. T., and Becker, B. (2005). Pamira - a parallel sat solver with knowledge sharing. In Abadir, M. S. and Wang, L.-C., editors, MTV, pages 29–36. IEEE Computer Society.

[29] Sheeran, M., Singh, S., and Stålmarck, G. (2000). Checking safety properties using induction and a sat-solver. In Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, FMCAD '00, pages 108–125, London, UK, UK. Springer-Verlag.

[30] Silva, J. P. M. and Sakallah, K. A. (1994). Efficient and robust test generation-based timing analysis. In ISCAS, pages 303–306.

[31] Silva, J. P. M. and Sakallah, K. A. (1996). GRASP: A New Search Algorithm for Satisfiability. In Proceedings of ICCAD, pages 220–227, San Jose, California, United States.

[32] Singer, D. and Monnet, A. (2008). JaCk-SAT: a new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In Proceedings of the 7th international conference on Parallel processing and applied mathematics, PPAM'07, pages 249–258, Berlin, Heidelberg. Springer-Verlag.

[33] Sinz, C. and et al (2008). SAT Race 2008. http://baldur.iti.uka.de/sat-race-2008/index.html. Accessed on May 2012.

[34] Velev, M. and Bryant, R. (2001). Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In Proceedings of the 38th Design Automation Conference (DAC '01), pages 226–231.

[35] Zhang, H. (1997). SATO: an efficient propositional prover. In Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI, pages 272–275.

[36] Zhang, H., Bonacina, M. P., Paola, M., Bonacina, and Hsiang, J. (1996). Psato: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation, 21:543–560.