UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JAIME KIRCH DA SILVEIRA

Parallel SAT Solvers and Their Application in Automatic Parallelization

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Luigi Carro
Advisor

Porto Alegre, march 2014

CIP – CATALOGING-IN-PUBLICATION

Jaime Kirch da Silveira,

Parallel SAT Solvers and Their Application in Automatic Parallelization /

Jaime Kirch da Silveira. – Porto Alegre: PPGC da UFRGS, 2014.

122 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2014. Advisor: Luigi Carro.

1. Parallel SAT Solver. 2. Automatic Parallelization. 3. RePaSAT. 4. SAT-PaDdlinG. I. Carro, Luigi. II. Título.

“The great end of life is not knowledge but action.”
— THOMAS HENRY HUXLEY

AGRADECIMENTOS

Apesar do resto deste documento estar em inglês, gostaria de apresentar meus agradecimentos em português, para garantir que todas as pessoas a quem eles se destinam possam compreendê-lo. Apesar de ficar diferente do resto do texto, é por uma causa nobre.

Em primeiro lugar, gostaria de agradecer à minha família. Agradeço à minha mãe, por sempre estar do meu lado e sempre me apoiar em todos os desafios que eu deseja encarar, por me dar apoio durante momentos difíceis. À minha irmã, Laura, que, apesar de distante, sempre esteve do meu lado, me deu apoio e acreditou em mim. Ao meu irmão, Marcelo, que sempre pôs fé na minha capacidade e me diz para sempre tentar ir longe. À minha irmã, Ingrid, que, sempre carinhosa, me traz uma grande sensação de apoio e segurança, assim como ao Maurício e ao meu querido sobrinho, Johann. Ao meu pai, que me apoia e acredita em mim. Ao Paulo, pelo apoio e amizade, por ter feito parte de nossas vidas por tanto tempo. À Cladir que ajudou a me criar e ainda faz parte das nossas vidas, sempre me deu muito apoio e carinho. A todos os demais dos meus familiares, minhas duas queridas tias, Gladis e Vera, ao meu tio, Sérgio, aos meus primos, Fernanda, Giovani, você continua nos nossos corações, Fernando, Rodrigo, ao meus avós, que sempre me mostraram muito carinho e qualquer outra pessoa da minha família que me ajudou e me apoio nesta jornada.

Em segundo lugar, gostaria de agradecer ao pessoal do laboratório, que me apoiou a atingir essa grande conquista. Ao grande mestre Vanius, pela grande ajuda (inclusive no Latex) e encorajamento. À minha grande amiga e colega de ônibus, Elen, pelo apoio e grande ajuda na organização, foram valorosas lições. Ao Diego, por fazer parte desse trabalho no início, pela ajuda e pela amizade. Ao Sérgio, pela amizade e apoio. Ao Carlos, ao Jacob, ao Thiago, ao Vinicius, todos participaram dessa conquista. A todos os amigos que conheci na UFRGS e me apoiaram e mostram amizade. Agradeço também ao meu orientador, Luigi, por ter acreditado em mim e me guiado nessa trajetória.

Esses dois anos foram anos de muita mudança, muita evolução e muitos desafios. A todos os amigos que conheci, que me apoiaram, que participaram da minha vida e que tornaram cada dia melhor, fica meu agradecimento. Aos grandes amigos e equipe do DL, que me ajudaram a crescer e acreditar em mim mesmo, ao meus amigos da academia, ao pessoal do meu prédio e todos aqueles que já passaram por mim, com um sorriso de amizade. A todos que tiveram um participação direto ou indireta nesse trabalho de 2 anos, seja me ajudando a formatar ou organizar o trabalho ou simplesmente acreditando em mim, tornando cada momento mais fácil, com amizade e carinho. Eu acredito que nossas conquistas são um pouco nossas e um pouco das pessoas que participam de nossas

vidas. Muito abrigado pelo grande apoio nessa jornada.

SAT Solvers Paralelos e Suas Aplicações em Paralelização Automática

RESUMO

Desde a diminuição da tendência de aumento na frequência de processadores, uma nova tendência surgiu para permitir que softwares tirem proveito de harwares mais rápidos: a paralelização. Contudo, diferente de aumentar a frequência de processadores, utilizar parallelização requer um tipo diferente de programação, a programação paralela, que é geralmente mais difícil que a programação sequencial comum. Neste contexto, a paralelização automática apareceu, permitindo que o software tire proveito do paralelismo sem a necessidade de programação paralela. Nós apresentamos aqui duas propostas: SAT-PaDdlinG e RePaSAT. SAT-PaDdlinG é um SAT Solver DPLL paralelo que roda em GPU, o que permite que RePaSAT utilize esse ambiente. RePaSAT é a nossa proposta de uma máquina paralela que utiliza o Problema SAT para paralelizar automaticamente código sequencial. Como uma GPU provê um ambiente barato e massivamente paralelo, SAT-PaDdlinG tem como objetivo prover esse paralelismo massivo a baixo custo para RePaSAT, como para qualquer outra ferramenta ou problema que utilize SAT Solvers.

Palavras-chave: SAT Solver Paralelo, paralelização automática, RePaSAT, SAT-PaDdlinG.

Parallel SAT Solvers and Their Application in Automatic Parallelization

ABSTRACT

Since the slowdown in improvement in the frequency of processors, a new tendency has arisen to allow software to take advantage of faster hardware: parallelization. However, different from increasing the frequency of processors, using parallelization requires a different kind of programming, parallel programming, which is usually harder than common sequential programming. In this context, automatic parallelization has arisen, allowing software to take advantage of parallelism without the need of parallel programming. We present here two proposals: SAT-PaDdlinG and RePaSAT. SAT-PaDdlinG is a parallel DPLL SAT Solver on GPU, which allows RePaSAT to use this environment. RePaSAT is our proposal of a parallel machine that uses the SAT Problem to automatically parallelize sequential code. Because GPU provides a cheap, massively parallel environment, SAT-PaDdlinG aims at providing this massive parallelism and low cost to RePaSAT, as well as to any other tool or problem that uses SAT Solvers.

Keywords: Parallel SAT Solver, Automatic Parallelization, RePaSAT, SAT-PaDdlinG.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic-Logic Unit
API	Application Programming Interface
BCP	Boolean Constraint Propagation
CDCL	Conflict-Driven Clause Learning
DLIS	Dynamic Largest Individual Sum
CNF	Conjuntive Normal Form
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DPLL	Davis-Putnam-Loveland-Logemann
GP-GPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
GRASP	Generic seaRch Algorithm for the Satisfiability Problem
RePaSAT	Reduction to Parallel SAT
SAT-PaDdlinG	SAT Parallel DPLL in GPU
SIMD	Simple-Instruction Multiple-Data
SIMT	Simple-Instruction Multiple-Thread
UIP	Unique Implication Point
VSIDS	Variable State Independent Decaying Sum

CONTENTS

1 INTRODUCTION

Since the beginning of computers history, there has been an attempt to increase the computational power available. In the 1980s, there was a trend of increasing the frequency of processors, guided by industry and academy, to provide faster processing (SILVA; BUYYA, 1999). For many year, it was successful, following Moore's law. However, this strategy seems to have reached its limit around 2003 (DIAZ; CARO; NINO, 2012) and it has become more and more difficult to further increase the frequency of processors, mainly due to heat dissipation and energy consumption (DIAZ; CARO; NINO, 2012). The solution found by industry has been to increase the number of cores, instead of frequency. Hence, parallelization is, nowadays, the main approach to increase the computational power, and it has become essential to achieve the necessary performance to computational needs.

Parallel programming is harder than common sequential programming, it is more error-prone and harder to debug and test. Developers must deal with problems that do not occur in sequential programming, such as: non-determinism, communication, synchronization, data partitioning and distribution, load balancing, fault tolerance, heterogeneity, shared or distributed memory, deadlocks and race conditions (BUYYA; SILVA, 1999). One must also consider that, given the new hardware platforms based on many cores, legacy code must be automatically parallelizable, so that software development costs can be contained. All these factors motivate the study of automatic parallelization of code.

There are several approaches available for automatic parallelization (AMBRUS, 2003; CORDES; MARWEDEL; MALLIK, 2010; PADBERG; MIROLD, 2012; BRADEL; ABDELRAHMAN, 2007). However, these tools are usually limited to some specific applications (SILVA; BUYYA, 1999), such as loop-based parallelism and vector libraries' utilities (SILVA; BUYYA, 1999; ARMSTRONG; EIGENMANN, 2008). Reference (AMBRUS, 2003) has shown that, for full industrial applications, common loop-based automatic parallelization is not able to achieve speedup compared to manual parallelization.

This document intends to present two ideas. The first, presented in Chapter 4, is the description of our implemented SAT Solver for GPU, called SAT-PaDdlinG. SAT

PaDdlinG is a DPLL SAT Solver for GPU, as a matter of fact, the first one. It was developed to allow a SAT Solver to take advantage of the massive parallelism brought by GPU environments, while still having most optimized algorithms found in literature for DPLL SAT Solvers. This SAT Solver was developed because RePaSAT needs a massive parallel environment, and GPU is a cheap one, that may bring the parallelism we need. SAT-PaDdlinG was initially created to be the SAT Solver behind RePaSAT. Nevertheless, it is a generic SAT Solver and may be used in many different contexts.

Our second proposal, presented in Chapter 5, is our proposal of a tool for automatic parallelization, called RePaSAT. This tool is capable of receiving an imperative and declarative code and automatically parallelizing it, by using a SAT Solver, a solver of the SAT Problem. Because parallel programming is generally harder than sequential programming, this tool allows the advantage of parallel processing, without parallel programming.

In regard to these two proposals, we study the following topics:

1. The viability of parallelism of a DPLL SAT Solver on a SIMT environment, such as a CUDA GPU, the environment SAT-PaDdlinG is built in.
2. The implementation of SAT-PaDdlinG, as the option for massive parallelism for RePaSAT.
3. Some GPU characteristics preventing SAT-PaDdlinG to parallelize on this GPU environment, as well as proposals to cope with them.
4. The translation of imperative specification of programs as CNF Boolean formulas (translation necessary for RePaSAT).
5. The improvement of the generated formulas' execution time by means of parallelism.
6. The parallelization of these formulas on GPU environment (SAT-PaDdlinG). Analysis on a CPU for comparison.

As for the other chapters, Chapter 2 shows a few basic concepts necessary to fully understand RePaSAT and SAT-PaDdlinG. The ideas behind SAT Solvers and CUDA are presented. Chapter 3 presents related works. Chapter 6 will present the most important contributions this work has provided and what advantages they bring with them as well as a conclusion of all this work. Chapter 7 presents future work for the ones presented here.

2 BASIC CONCEPTS

This chapter presents some basic concepts. It presents the whole idea behind SAT Solvers, including how they work and what strategies are used to improve their efficiency. Because SAT Solvers, in special parallel SAT Solvers, are the idea behind RePaSAT and SAT-PaDdlinG, these concepts are necessary for fully understanding those two ideas.

We also present here the basic concepts about CUDA, the technology over which SAT-PaDdlinG was developed. We present its objective, its basic concepts and a little of history.

This chapter is organized as follows: Section 2.1 presents the basic concepts of the SAT Problem and SAT Solvers. Section 2.2 presents a specific kind of SAT Solvers, the DPLL SAT Solvers, the one we focus on this document. Section 2.3 presents the main optimizations for DPLL SAT Solvers found in literature. Section 2.4 presents parallel SAT Solvers, their characteristics and challenges. And, at last, Section 2.5 presents some characteristics of the CUDA architecture, over which SAT-PaDdlinG is build.

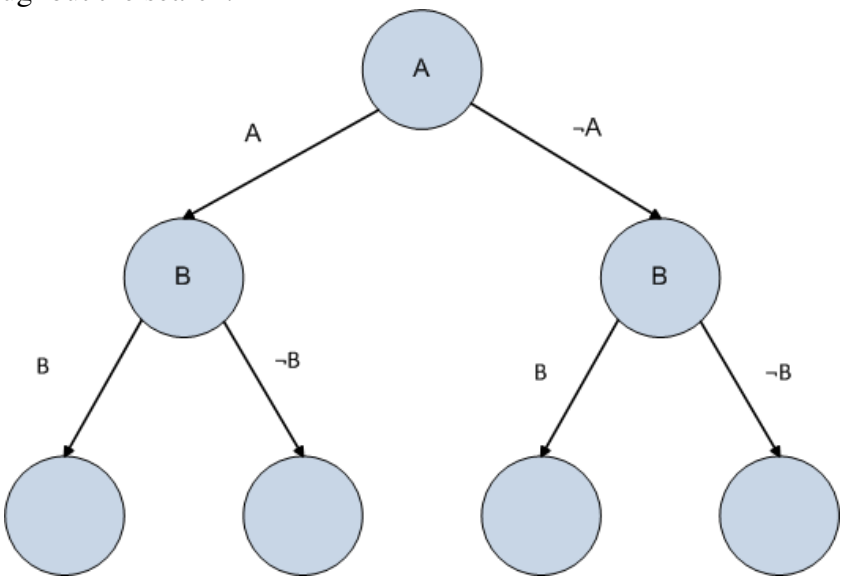
2.1 The SAT Problem

SAT Solvers are software developed to solve the SAT problem. This problem is a known NP-Complete problem (BENHAMOU et al., 2010). Indeed, it was the first decision problem to be proven NP-complete (BENHAMOU et al., 2010). This characteristic implicates that this problem has greater than polynomial-time complexity in the worst case, a scenario that can only change if $NP = P$.

However, because of its large applicability in several areas, including automatic deduction, configuration, planning and scheduling (BENHAMOU et al., 2010), many algorithms were introduced to cope with its exponential-time. Modern SAT Solvers are, nowadays, able to solver in viable time some instances of problems with hundreds of thousands of variables (HAMADI; JABBOUR; SAIS, 2009), by using a set of techniques.

Modern parallel SAT Solvers also implement such techniques. Although paral-

Figure 2.1: A search tree is a representation of the search the decisions made by the SAT Solver throughout the search.



lelism decreases the search time, it is not enough to cope with the exponential complexity of SAT Solvers on its own, since it can only reduce the processing time by a constant factor, the number of processing units. Thus, the optimized sequential algorithms are still present on parallel SAT Solvers, and their adaptation to a parallel paradigm is often necessary.

The SAT problem is the decision problem of determining if a Boolean formula is satisfiable or not (KROENING; STRICHMAN, 2008). As a decision problem, the expected output for any input is either SAT (is satisfiable) or UNSAT (it is not satisfiable). A satisfiable formula F has a Boolean assignment ('true' or 'false') to all of its variables such that it satisfies F (is interpreted as true) (KROENING; STRICHMAN, 2008). This formula has a set of Boolean variables and Boolean operations connecting them. Determining whether a formula is satisfiable or not is seldom enough. Therefore, SAT Solvers also provide an assignment to every variable that satisfies the formula (KROENING; STRICHMAN, 2008), in case it is satisfiable.

The SAT problem is generally solved through a search in the space of possible solutions. This search can be seen as a binary tree (ZHANG; BONACINA; HSIANG, 1996) in which each node refers to a variable and the two trees below the node are the rest of the search-space when assigning true and when assign false to that variable. In a full search, when every possible value is assigned to every variable, we have a tree with height equal to the number of variables, and with 2^n nodes. Each branch of the tree refers to one assignment of the variables. This tree is often referred as search tree (ZHANG; BONACINA; HSIANG, 1996).

Figure 2.1 illustrates the search in a formula with two variables, A and B. It is a

full search, since every value to every variable has been assigned. The leaf nodes represent states in which all variables are assigned.

Most SAT Solvers solve formulas in CNF (KROENING; STRICHMAN, 2008). A formula in this form uses only the operations: the conjunction or logic 'and' ($\wedge$), the disjunction or logic 'or' ($\vee$) and logic 'not' ($\neg$). A formula in CNF is a conjunction of clauses (KROENING; STRICHMAN, 2008). A clause is a disjunction of literals (KROENING; STRICHMAN, 2008). Literals are either a variable or a negated variable (KROENING; STRICHMAN, 2008). The formula 2.1 is in CNF.

$$(V_1 \vee V_2 \vee \neg V_3) \wedge (\neg V_1 \vee \neg V_2) \wedge (V_2 \vee V_3 \vee \neg V_4) \tag{2.1}$$

An important characteristic of CNF is that every Boolean formula can be translated into it in linear-time (KROENING; STRICHMAN, 2008), with only a linear increase in size (KROENING; STRICHMAN, 2008). It also allows us to reinterpret the problem. Now, we want to find assignments that make every clause true, considering that to make a clause true it is necessary to have at least one true literal in it. So we want assignments that cause at least on literal in each clause to be true.

An alternative version of the SAT problem is the 3-SAT problem. It is also an NP-Complete problem (KLEINBERG; TARDOS, 2006), so any SAT formula can be reduced to a 3-SAT formula in polynomial time. The 3-SAT Problem solves formulas in which all clauses have 3 literals (KLEINBERG; TARDOS, 2006). This size limitation may be useful in some contexts.

2.2 DPLL SAT Solvers

Most SAT Solvers can be divided into two categories: DPLL (Davis-Putnam-Loveland-Logemann) and stochastic search (KROENING; STRICHMAN, 2008). Since most stochastic searches are not complete (KROENING; STRICHMAN, 2008) and, thus, may not find a satisfiable assignment for satisfiable formulas, they are out of the scope of this document. DPLL are widely used, even in parallel SAT Solvers.

The DPLL procedure is a search through the search space in an attempt to find an assignment that satisfies the formula. The search is done through levels, starting on level 1 (KROENING; STRICHMAN, 2008). For every level, a value is assigned to a variable (KROENING; STRICHMAN, 2008), which is called a decision and may cause propagated assignments (HOLLDOBLER; MANTHEY, 2011), or implications. While no conflict or satisfiable assignment is found, the level is increased and new assignments are made. At any point of the search process, the SAT Solver has assigned variables (decisions

and implications) and free variables, to be assigned later. If the set of free variables is empty, all variables are assigned and we call it a complete interpretation (BENHAMOU et al., 2010), otherwise we have a partial interpretation (BENHAMOU et al., 2010).

If the current assignment makes the formula unsatisfiable, a conflict happens and the procedure identifies a level to backtrack to and carries out this backtracking, undoing every assignment made after that level (KROENING; STRICHMAN, 2008). This requires storing the decision level of each decision and implication, to know when to remove them. The execution resumes from the backtracked level (KROENING; STRICHMAN, 2008). If a satisfiable assignment is found, the procedures halts and returns it as the result. If it is backtracked to the decision level 0 and no satisfiable assignment is found, the procedure returns unsatisfiable (KROENING; STRICHMAN, 2008).

To analyze a conflict, the SAT Solver uses a conflict graph (KROENING; STRICHMAN, 2008). It is a DAG that represents every assignment made up to the current level, as well the decision level each assignment was made (KROENING; STRICHMAN, 2008). With this graph, it is possible to identify a conflict, when it happens, and to determine to which level the procedure should backtrack to. It also allows the SAT Solver to learn a clause. This graph is further described in Section 2.3.1.

2.2.1 Basic Concepts

As mentioned before, formulas in CNF can be seen as actually a set of clauses. Indeed some papers refer to them as clauses databases. This characteristic brings a lot of gains. For starters, we can redescribe the problem as the problem to have every clause satisfied. That breaks the problem into separate parts that we want to satisfied, although one affect the other. Besides, only by having this well defined portion of the formula, can we learn new clauses, a concept mentioned in Section 2.3.1. At any moment, under a partial (or complete) interpretation, a clause may be in one of these states:

Satisfied : At least one of its literal is evaluated true (KROENING; STRICHMAN, 2008) (its variable was assigned the same polarity as its literal in the clause).

Conflicting : All of its literals are evaluated false (KROENING; STRICHMAN, 2008). Not necessarily the variables are false, but their assignments make the literals in the clause false.

Unit : All but one literal in this clause are evaluated false (KROENING; STRICHMAN, 2008), the remaining one is free (unassigned) (KROENING; STRICHMAN, 2008). Unit clauses are the basic idea behind BCP, which allows implications from decisions.

Unresolved : Any other state (KROENING; STRICHMAN, 2008). Basically, a clause containing at least 2 unassigned literals and no satisfied one.

Do not forget that these states depend on the current assignments, on the current decisions and implications. Once these assignments change, the clauses states may change as well. Any unit clause C will result in a implication l, where l is the only unresolved literal of C, since l must be true not to falsify this clause. In this case, we say that C is the antecedent clause of l (KROENING; STRICHMAN, 2008). Equations from 2.2 to 2.6 show an example of clauses states. Equation 2.2 shows assignments, while Equations from 2.3 to 2.6 show clauses and their states given those assignments.

$$Assignments : \{a := T, b := T, c := F\} \tag{2.2}$$

$$(a \vee c) : (satisfied) \tag{2.3}$$

$$(\neg a \vee \neg b \vee c) : (conflicting) \tag{2.4}$$

$$(\neg a \vee \neg b \vee d) : (unit) \tag{2.5}$$

$$(\neg a \vee d \vee e) : (unresolved) \tag{2.6}$$

A polarity is either the value $true$ or $false$ assigned to a variable (KROENING; STRICHMAN, 2008). A decision or an implication is an assignment of polarity to a variable. Since a literal is a variable with a polarity (in a clause), sometimes assignments like decisions and implications are referred as literals. For example, assignment $false$ to variable a may be simply expressed as the literal $\neg a$. Some SAT Solvers, such as MiniSAT (EÉN; SÖRENSSON, 2004), also have the concept of assumptions, which are like decisions made before the search starts. These assumptions are never undone during the search and are always considered true. The SAT Solvers look for a solution that also satisfies the assumptions, as though the assumptions were part of the formula.

2.2.2 Basic Procedures

The algorithm for DPLL search is depicted in Algorithm 1. During the search, a DPLL SAT Solver carries out the following procedures:

Algorithm 1: Basic algorithm from a DPLL SAT Solver.

```
Data: A CNF formula F and three sets: decisions (empty), implications (empty)
      and free_variables containing all variables
Result: Whether F is satisfiable or not
while not free_variables empty do
    branch() ;                                  /* Makes a decision */
    BCP() ;                                /* Propagates the decision */
    if conflict then
        backtracking_level = analyze_conflict();
        if backtracking_level == 0 then
            return UNSAT;
        end
        backtrack(backtracking_level);
    end
end
return SAT;
```

Branching : Also known as decision procedure. This procedure decides a variable to branch the search space in, as well as a polarity (KROENING; STRICHMAN, 2008). Once a decision is made, the decision level is incremented (KROENING; STRICHMAN, 2008), allowing only one decision per level, though there may be many implications per decision (which have the same decision level as their implicating decisions). There are many heuristics that attempt to make a decision that will lead faster to the answer, such as VSIDS (KROENING; STRICHMAN, 2008). Once we select a decision, we mark it as branched once (only one of its possible values was tested) to know we still need to test the other assignment.

BCP : After a decision is made, it must be analyzed to determined two things: the presence of implications from the decisions and conflicts. Implications are found through the unit rule (KROENING; STRICHMAN, 2008), which simply states that, for every unit clause, the only free literal implicates to its variable its polarity (e.g. in a unit clause where only $\neg A$ is free, it implicates $false$ to A). Implications may cause new implications (cause new unit clauses), thus it is necessary to propagate every new implication, possibly revisiting clauses. A conflict is found if there is at least one conflicting clause. That situation is only possible through implications and it results in backtracking.

Conflict analysis. Once a conflict is found, it is analyzed to generate a learnt clause from it and determine a backtracking level. The conflict analysis is carried out with an implication graph, particularly by CDCL SAT Solvers and better explained in Section 2.3.1.1.

Backtracking : After a conflict is found and analyzed, the SAT Solver must re-

turn some levels, backtrack, undoing conflicting decisions. It may simply backtrack to the previous decision, in which case it is called a chronological backtracking (SILVA; SAKALLAH, 1997) (it may jump more than one level, if the previous level was already tested both ways). The SAT Solver can also carry out a non-chronological backtracking, idea presented by (SILVA; SAKALLAH, 1997) and largely used. In such case, the SAT Solver is capable of identifying a decision level before the previous decision level to backtrack to (SILVA; SAKALLAH, 1997). The decision of the backtracked is re-branched and the SAT Solvers continues.

During the execution of a SAT Solver, it starts with a decision (branching). After branching, it runs BCP to implicate and search for conflicts. If no conflict is found, the solver tests if there are no more free variables, in such case the SAT Solvers returns 'satisfiable'. If a conflict is found, it is analyzed and the SAT Solvers backtracks and flips the decision polarity of the backtracked level. The SAT Solvers goes back to BCP, for there may be other implications and another conflict, and keeps running BCP, analyzing and backtracking until there is no more conflict. If the SAT Solver backtracks to level 0, the problem is unsatisfiable. Otherwise, the SAT Solver does a new decision. The Algorithm 1 illustrates.

2.3 Main Optimizations of DPLL SAT Solvers

In this section we present the main optimizations for DPLL SAT Solvers. These optimizations are widely mentioned in literature and have greatly increased the efficiency of SAT Solvers. It is important to point out that many other optimizations are available and even the ones presented here may be differently implemented by different SAT Solvers. In those implementations, they may diverge a little from how they are described here, but the idea remains.

2.3.1 CDCL SAT Solvers

One of the characteristics that make current DPLL SAT Solver extremely efficient, in spite of its exponential-worst time complexity, is the ability to learn from mistakes, from conflicts. As mentioned before, during the search, the SAT Solver may identify a conflict. When a conflict is detected, we know that the partial assignments made so far are enough to make the problem unsatisfiable and we can discard the entire section of the search tree below this point, this section is called a no-good (GOMES et al., 2008).

Once a conflict is found, the SAT Solver is capable of identifying what caused the conflict and generating and learning a clause describing this conflict (KATEBI; SAKALLAH; MARQUES-SILVA, 2011). This learnt clause contains a number of negated (with the

opposite polarity) literals from the current decisions and/or the implication literals that, alone, cause the conflict (KATEBI; SAKALLAH; MARQUES-SILVA, 2011). Since most if not all current DPLL SAT Solver learn clauses, sometimes the concept of clause learning through conflict is described as part of the DPLL algorithm.

Though this learnt clause is generated through the current decisions and implications, it is actually a sole implication of the formula itself, for it will always be true, given the current formula, since its a consequence of the input (HOLLDOBLER; MANTHEY, 2011). Thus, it can be learnt and kept during the entire solving process (or delete, depending on the policy). Because this clause describes a conflict and has fewer literals than decisions and implications when the conflict occurred, it helps identify similar conflicts earlier in the process, finding no-goods that would not be identify as early and speeding up the solving. SAT Solvers that learn clauses from conflicts are called CDCL SAT Solvers. This idea was first proposed by GRASP (SILVA; SAKALLAH, 1997). To generate a clause from the conflict, SAT Solvers use an implication graph.

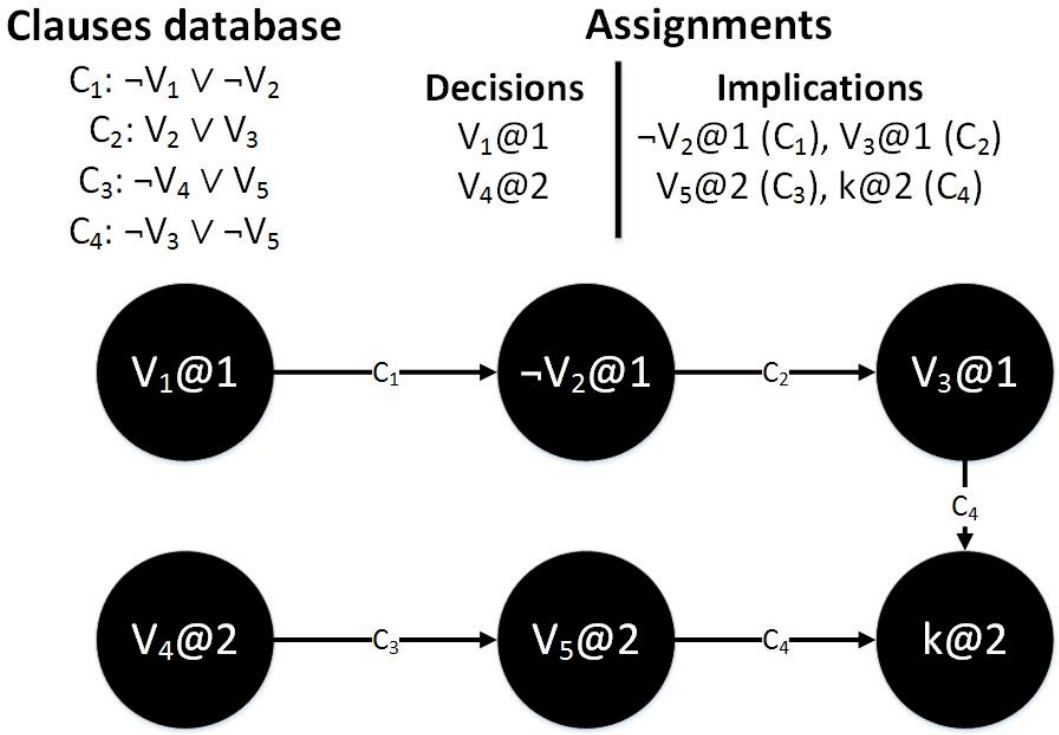
Implication graph. Implication graphs are dynamic structures, generated from the current decisions and implications (SILVA; SAKALLAH, 1997) and modified as new decisions are made and implications are generated, as well as when a backtracking is carried out. They contain a vertex for every decision and implication (SILVA; SAKALLAH, 1997) and the implications have an incoming edge from the decisions and/or implications that caused them (their antecedent assignment) (SILVA; SAKALLAH, 1997). Generally, when a conflict happens, a conflict vertex is added to the graph and its antecedent assignments are the literals of the conflicting clause. Figure 2.2 illustrates it with an example.

In Figure 2.2, we have an example of an implication graph. The formula contains the clauses in the database. In assignments, we have the current decisions and their implications and the number after the @ sign indicates in which decision level each decision and implication was made. The implication graph is depicted, each vertex contains an assignment. The edges contain the clause that caused the implications, the k symbol indicates a conflict.

In the first decision level, the decision is the assignment true to the variable V_1, which implicates in $\neg V_2$ and this implication implicates in V_3. In the second decision level, the assignment true to V_4 is done, which implicates in V_5, which, along with the V_3 implication of the previous decision level, results in a conflict, marked as k.

Notice that the decisions do not have antecedents and the implications have as antecedents the decisions or implications that implicated them. In this example, each implication only has one antecedent, except for the conflict, but this is not necessary always true. An implication has as many antecedent assignments as the number of literals in its antecedent clause minus 1. The conflict vertex's antecedents are the implications

Figure 2.2: An example of an implication graph.



that caused the clause C_4 to be conflicting.

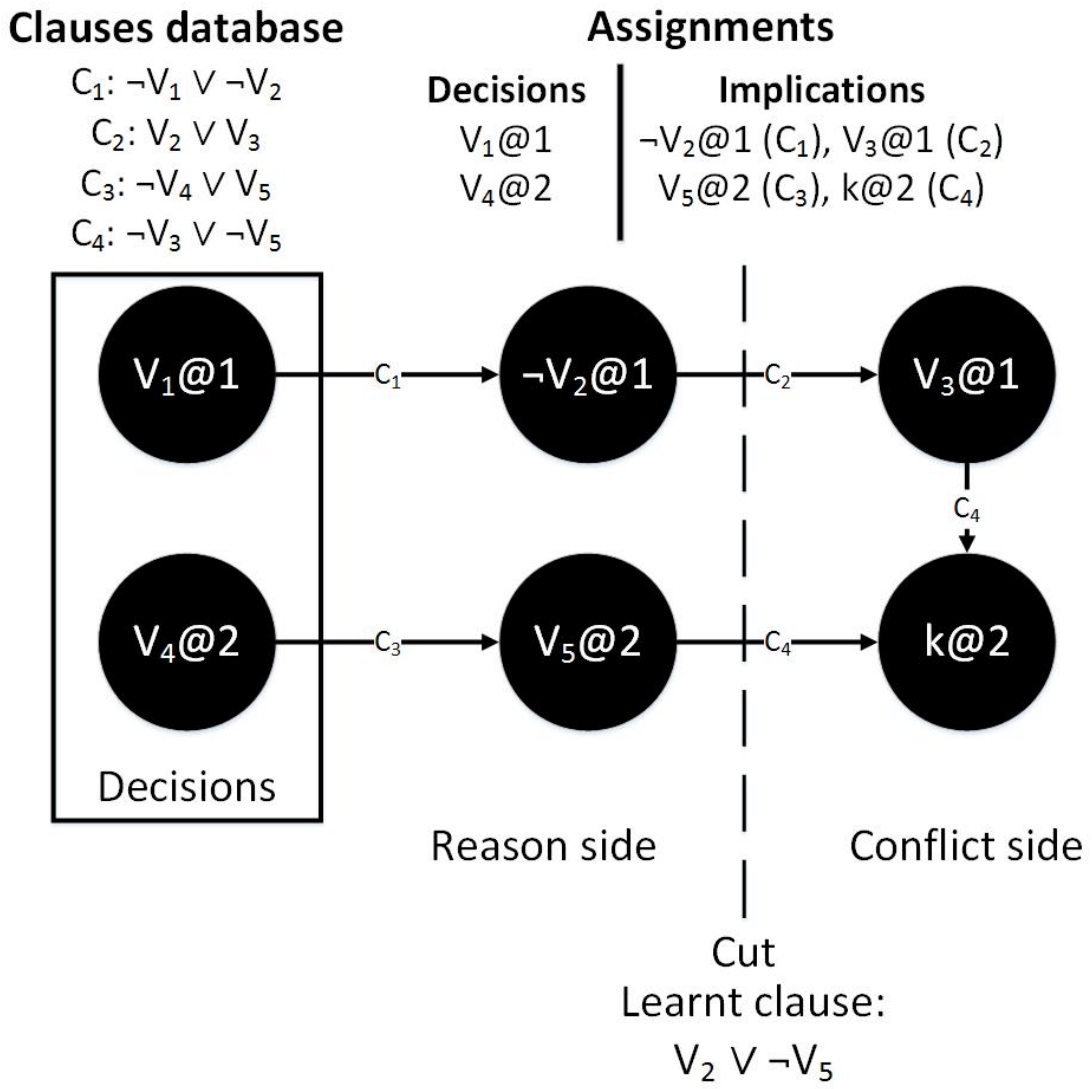
2.3.1.1 Learning from the Implication Graph

Once a conflict is found, the SAT Solver analyzes the conflicting implication graph to generate the clause (SILVA; SAKALLAH, 1997). The learnt clause is generated from any edge-cut in the graph that separates the decisions (reason side) from the conflict (conflict side) (GOMES et al., 2008). Figure 2.3 illustrates.

The generated learnt clause is the negation of the literals of the source of every edge in the cut (GOMES et al., 2008), in the example of Figure 2.3, it is $v_2 \vee \neg v_5$. Since there are many possible cuts, finding a cut that better improves the behavior of the SAT Solver is necessary. Nowadays, cuts using first UIP (KROENING; STRICHMAN, 2008) are very common. Most modern SAT Solvers generate asserting clauses as learnt clauses.

Asserting clause. The generated clause from the conflict may be an asserting clause. This special kind of clause causes an immediate implication after backtracking (KROENING; STRICHMAN, 2008). This clause can immediately implicate, because it becomes unit after backtracking (MADIGAN; MOSKEWICZ; MALIK, 2001), since all its literals but one have lower decision levels than the current one (PIPATSRISAWAT; DARWICHE, 2009) (before backtracking). This only literal that has the same decision level as the current, the one that can be learnt, is called the asserting literal (PIPATSRISAWAT; DARWICHE, 2009). Every conflicting state has an asserting clause that can be

Figure 2.3: An example of an implication graph with a cut separating the reason side to the conflict side.



learnt (PIPATSRISAWAT; DARWICHE, 2009).

2.3.2 Non-Chronological Backtracking

Non-chronological backtracking was introduced by GRASP (SILVA; SAKALLAH, 1997) to SAT Solvers, though already used in other areas. It is the capability of a SAT Solver to backtrack more than one level once a conflict is found (SILVA; SAKALLAH, 1997). When a conflict is detected, the SAT Solver must backtrack to a previous level. Backtracking to the previous level is trivial, since we know at least one decision must be undone. However, some SAT Solvers are capable or identifying decision levels further back to which the SAT Solver can backtrack.

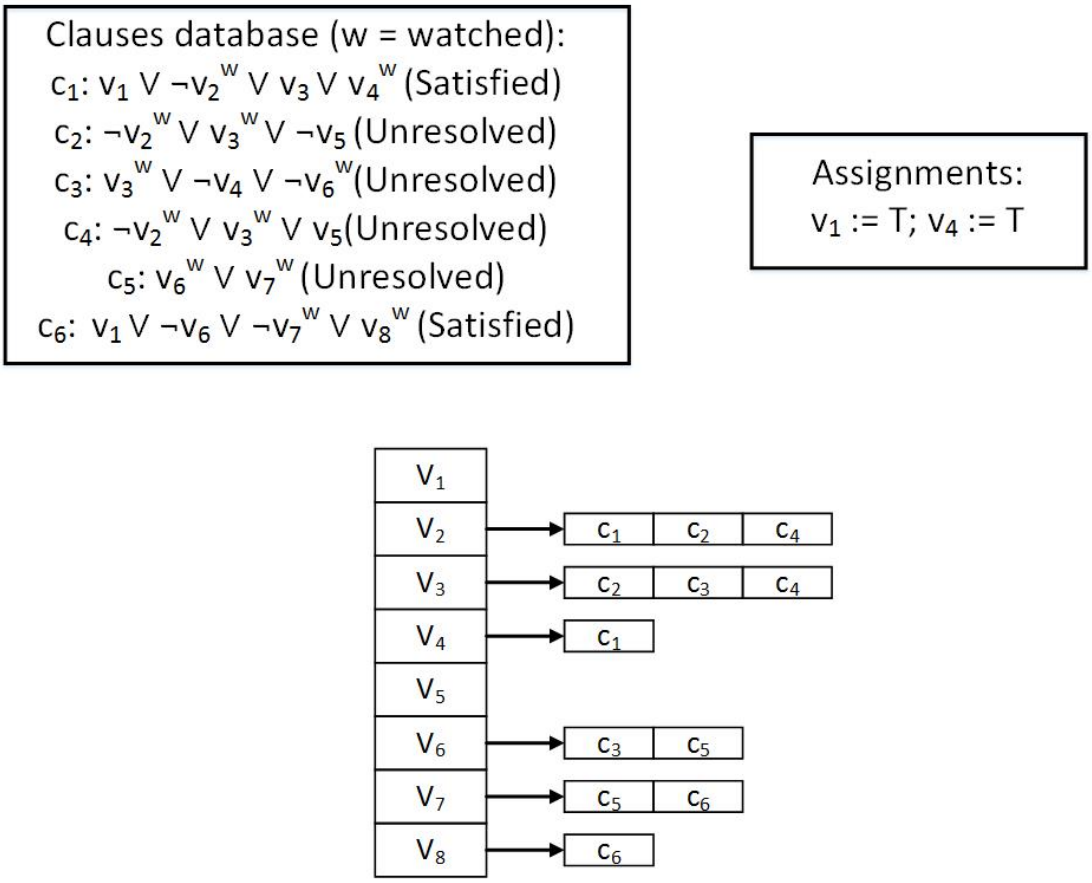
GRASP (SILVA; SAKALLAH, 1997) presents a non-chronological backtracking scheme that requires a second conflict to allow non-chronological backtracking (SILVA; SAKALLAH, 1997). In this scenario, a conflict happens with an asserting clause. Since backtracking directly results in an implication it may happen a second conflict. In this conflict, the generated clause may have the decision levels for all its literals lower than the current (SILVA; SAKALLAH, 1997). In this case, the SAT Solver backtracks to the highest of these levels, carrying out a non-chronological backtracking (SILVA; SAKALLAH, 1997).

However, backtracking has been refined since GRASP (MARQUES-SILVA; LYNCE; MALIK, 2009). Chaff (MOSKEWICZ; MADIGAN; ZHAO, 2001) proposed to always backtrack to the level of the second highest literal (MARQUES-SILVA; LYNCE; MALIK, 2009). Because the learning clause is an asserting clause, even when backtracking to the level of the literal decided in the second highest literal, this clause will remain unit (MARQUES-SILVA; LYNCE; MALIK, 2009) and will cause an implication. In this scheme, completeness is still guaranteed (MARQUES-SILVA; LYNCE; MALIK, 2009; KROENING; STRICHMAN, 2008). This results in significantly more efficient backtrackings (MARQUES-SILVA; LYNCE; MALIK, 2009) and empirical evidence shows that this strategy along with conflict-driven decision heuristics performs very well (MARQUES-SILVA; LYNCE; MALIK, 2009).

2.3.3 Two-Watched Literals

In a DPLL SAT Solver, the BCP procedure is the most time-consuming one, consuming more than 90% in most cases (MOSKEWICZ; MADIGAN; ZHAO, 2001; SINGER, 2006). Thus, improving this process may save a lot of time of analysis. In this regard, the two-watched literals strategy was created. The two watched-literal strategy speeds up BCP, by 'indexing' the clauses, with regard to two watched literals (MOSKEWICZ; MADIGAN; ZHAO, 2001).

Figure 2.4: An example of two watched literals, given some assignments.



Because BCP attempts to find either unit clauses or a conflict, it only has to search for clauses with at most one free literal (MOSKEWICZ; MADIGAN; ZHAO, 2001). As a matter of fact, it only has to look for clauses with at most 1 literal not unsatisfied (consistent) (MOSKEWICZ; MADIGAN; ZHAO, 2001), because unit clauses have only 1 consistent literal, while a conflicting clause has none.

To take advantage from that, the two-watched literals scheme keeps two watched literals for every clause and, when a decision or implication is made, it only searches through clauses whose watched literals contain this assignment (MOSKEWICZ; MADIGAN; ZHAO, 2001). To accomplish that, they keep a list for every variable containing a pointer to each clause that watches a literal of that variable (MOSKEWICZ; MADIGAN; ZHAO, 2001). For every decision or implication, only those clauses that watch the variable are analyzed (MOSKEWICZ; MADIGAN; ZHAO, 2001), removing a large portion of clauses. Since the clause has two consistent literals, to become conflicting or unit, it must contain at most 2 literals and the assigned variable must be one of them, assuring that the implication or conflict will always be found (MOSKEWICZ; MADIGAN; ZHAO, 2001). The Figure 2.4 illustrates.

Once a clause is analyzed after an assignment, three possible situations may occur

(MOSKEWICZ; MADIGAN; ZHAO, 2001):

- There are still at least two consistent literals, in which case the watched literals are modified to be consistent. No implication or conflict is detected in this clause. In this case, if the previously watched literals have become unsatisfied, they must be unwatched and other literals, which must be consistent, must be watched.

- There is only one consistent literal (always the other watched one). In this case, this literal is returned, if it is free, as an implication of the current assignment. This implication must be propagated in the two-watched literals structure.

- There is no consistent literal in the clause. In this case, the clause is returned as a conflicting clause. A conflict has been detected.

This strategy is used by many common SAT Solvers, such as the competition-winner MiniSAT (EÉN; SÖRENSSON, 2004) and Chaff (MOSKEWICZ; MADIGAN; ZHAO, 2001). Although this strategy is not capable of improving the BCP's worst-case complexity, since in the worst case it will have to search through every clause, it generally makes the BCP search through much fewer clauses than the total amount of them. For a clause to be visited with two-watched literals scheme is must contain the assignment variable and be watched, which greatly reduces the probability of a visit.

2.3.4 Restarts

Restarts are strategies created to cope with a problem called the heavy-tailed cost distribution (GOMES; SELMAN; CRATO, 1997), that could cause exponential behavior (KATEBI; SAKALLAH; MARQUES-SILVA, 2011). To escape from regions of the problem that do not contain the solution (KATEBI; SAKALLAH; MARQUES-SILVA, 2011), current SAT Solvers use restarts. When restarted, a SAT Solver goes back to the first decision level (KATEBI; SAKALLAH; MARQUES-SILVA, 2011). It undoes all decisions and implications, but keeps other data that help with the search, such as learnt clauses (HUANG, 2007).

Because SAT Solvers keep learnt clauses after restarting, that prevents them from searching through search-spaces previously discarded, discarding them by conflicts (HUANG, 2007). They also allow the SAT Solver to make better decisions in the beginning of the search (HUANG, 2007). The first decisions are important and since the SAT Solver, after restarting, has kept every data, including for possible dynamic decision heuristics, such as VSIDS, it may make smarter decisions in the beginning (HUANG, 2007). As (HUANG, 2007) puts it, during the search the SAT Solver develops beliefs about decisions to make, but is bounded to their previous decisions, made without the benefit of this knowledge.

For that reason, (HUANG, 2007) states that it is probably the reason restarts improve the efficiency of SAT Solvers, even without randomness.

If a SAT Solver deletes their learnt clauses and carries out restarts, it is not certain to finish solving (HUANG, 2007), since it does not keep all searched paths and may repeat the same over and over again. This is a common behavior in SAT Solvers, since the clause database may become very big after some time. One approach to solve this problem, used by zChaff (MOSKEWICZ; MADIGAN; ZHAO, 2001) and other SAT Solvers is to keeps decreasing the frequency of restarts, extending the time between each restart. This guarantees that eventually the SAT Solver will find the answer, since at some point, the SAT Solver will have enough time to find the answer before restarting.

There are many restarts policies and (HUANG, 2007) presents some of them. These policies include the Loby restart, which defines a number N of conflicts and uses the sequence: $N, N, 2\times N, N, N, 2\times N, 4\times N, N, N, 2\times N, N, N, 2\times N, 4\times N, 8\times N...$ to define after how many conflicts the SAT Solver must restart (HUANG, 2007). There are geometric policies that use N conflicts and p portion of increase ($p > 1$). They begin by restarting after N conflicts and then update to $N = N * p$, increasing N geometrically. In (HUANG, 2007), two examples are presented, one used by MiniSAT v1.14, with $N = 100$ and $p = 1.5$ and the other with $N = 32$ and $p = 1.1$, created by (HUANG, 2007).

2.3.5 Decision Heuristics

As the SAT Solver keeps increasing the decision level, it must make new decisions. These decisions deeply affect the behavior of the SAT solver. In (KROENING; STRICHMAN, 2008), decisions heuristics are described as probably the most important in SAT Solving. For that reason, many heuristics have come up to improve how SAT Solvers make decisions.

One of the most common decision heuristics is VSIDS. VSIDS is a dynamic heuristic that keeps a sum for every possible literal (KROENING; STRICHMAN, 2008), which are initialized to 0 (SHACHAM; ZARPAS, 2003). When a decision is to be made, the free literal with the highest sum is chosen (SHACHAM; ZARPAS, 2003), thus it also choses the polarity of the decision. The sum is update every time a learnt clause is generated (SHACHAM; ZARPAS, 2003), that is, it is a conflict-driven heuristic (KROENING; STRICHMAN, 2008). Once a clause is learnt (SHACHAM; ZARPAS, 2003), all its literals' sum are incremented (SHACHAM; ZARPAS, 2003). Periodically, the sums of all literals are divided by a constant (SHACHAM; ZARPAS, 2003) to give more importance to earlier conflicts (KROENING; STRICHMAN, 2008).

Another commonly mentioned heuristic is DLIS. This algorithm simply assigns a value to an unassigned variable in such way that it satisfies the most so far unsatis-

fied ones (KROENING; STRICHMAN, 2008). Generally, to accomplish that, the SAT Solvers keeps a pointer to the clauses that contain a literal and it counts the number of unsatisfied clauses that contain the literal (KROENING; STRICHMAN, 2008). The literal with the highest count is chosen (KROENING; STRICHMAN, 2008). This imposes a large overhead (KROENING; STRICHMAN, 2008), since it grows with the number of clauses (KROENING; STRICHMAN, 2008). However, it is understandable why such heuristic would be used. Since we want to satisfy all clauses, this choice may tend to lead us to the right solution.

In DLIS, the decision made is the one the satisfies the most clauses (KROENING; STRICHMAN, 2008). Since the SAT Solver's ultimate goal is to satisfy all clauses, this heuristic helps direct the SAT Solver to this goal.

Other possible approach is to use the RAND heuristic. This is a simple approach where a random unassigned variable is chosen (MOSKEWICZ; MADIGAN; ZHAO, 2001). This approach is both easy to implement and does not bring with it much overhead (almost none). However, it is trivial to see that RAND does not take advantage of learning and does not improve its behavior over time.

The Bermin heuristic keeps a sum to every literal, as VSIDS does, increasing when new clauses are learnt (KROENING; STRICHMAN, 2008), but it does not divide them by the constant. It keeps the learnt clauses and when needs to make a decision, finds the last learnt clause that is still unresolved (KROENING; STRICHMAN, 2008). From the literals is this clause, the decision is the literal with the largest sum (KROENING; STRICHMAN, 2008).

2.4 Parallel SAT Solvers

Many optimizations have made SAT Solvers faster and able to solve previously unsolvable instances. However, it is still a challenge to solve some instances, that seem to be very hard. To further improve the efficiency of SAT Solvers and allow them to solve even more instances, parallel SAT Solvers have been developed. Parallel SAT Solvers have the advantage of being able to take advantage of parallel environment, cutting short the execution time. Because parallelization has become a tendency, such approaches are necessary to take advantage of modern hardware.

However, as with any other parallel application, parallelizing SAT Solvers brings a whole new set a problems to be coped with. Such problems are most related to communication and work sharing. This section aims to show the basic challenges of parallel SAT Solvers, as well as strategies to cope with them.

2.4.1 Workload Distribution Problem

It is not always easy to divide the problem to be solved in parallel. The most obvious way to parallelize the problem is by dividing the search space (by using a divide-and-conquer strategy). This approach consists in dividing the formula in many sub-formulas to be solved separately. However, it is hard to determine the relevant variables to divide the search space with (HAMADI; JABBOUR; SAIS, 2009).

Simply dividing the search space in one part for each processor may result in a poor distribution of the problem, possibly with one processor doing most of the work. Indeed, (MARTINS; MANQUINHO; LYNCE, 2010) states that a drawback of search space splitting is load balancing. That happens because it is hard to predict the time needed to complete a specific branch of the search tree (SINGER, 2006; MARTINS; MANQUINHO; LYNCE, 2010), because the distributions of solutions in non-uniform in average (SINGER, 2006) and so is the time to solve a sub-formula (JURKOWIAK; LI; UTARD, 2005).

Thus, it is difficult to statically partition the search space in the beginning of the algorithms (SINGER, 2006) and guarantee a good distribution of the search space. Hence, the search space is typically dynamically divided (SINGER, 2006; MARTINS; MANQUINHO; LYNCE, 2010) and the work is distributed to the processors in runtime (SINGER, 2006).

A big challenge is to balance the workload, reducing the idle time of the processors, but also reducing the balancing process computing and communication time (SINGER, 2006). On one hand, distributing large parts of the problem to each processor may negatively affect the balance of workload, as it becomes harder to predict how long each part will take to be completely searched. On the other hand, distributing small portions of the problem may result in too much communication and balance processing for only small processing time to solve the specific part.

Determining the amount of workload to be distributed is an important problem to be solved. A situation in which to little work is given to each processor may cause “The Ping-Pong Phenomenon” (JURKOWIAK; LI; UTARD, 2005). This describes a situation in which the processes spend more time in communication than in processing, so data is sent from one process to another like a ping-pong ball (JURKOWIAK; LI; UTARD, 2005).

2.4.2 Parallelization Strategies

By parallelization strategies, we refer to the strategies the SAT Solvers use to have many processing units solve the same problem, in such way that it becomes faster than

sequential. These strategies are a very important issue in parallelism. If badly handled, they may cause processes to be idle or to generate too much communication, which may greatly decrease the expected efficiency. There are many strategy used in literature to distribute the problem between processing units. We show them divide in two groups: Cooperative Parallel SAT Solvers and Competitive Parallel SAT Solvers (IRFAN; MANTHEY, 2013).

2.4.2.1 Cooperative Parallel SAT Solvers

Cooperative Parallel SAT Solvers attempt to split the search-space in different parts (IRFAN; MANTHEY, 2013), in order to give each processing unit a part to process. Thus, it allows each thread to search through a different part of the search-space, reducing the search for each thread. Because they cooperate, that is, they divide their search in a cooperative manner, they are called a cooperative approach. These approaches are often referred to as "divide-and-conquer" strategies. We show here three different cooperative search strategies: Static Divide-and-Conquer Strategy, Task Farm and Dynamic Workload Stealing.

2.4.2.1.1 Static Divide-and-Conquer Space

The Divide-and-Conquer Space (HAMADI; JABBOUR; SAIS, 2009) is a strategy to divide the workload. It consists in assigning different values to variables by different processes (KOTTLER; KAUFMANN, 2011). For example, if two process are meant to solve a formula F, a variable v is chosen. The first process assigns the value true to v and solves the resulting formula with this assignment. Analogously, the second process assigns false to v and solves the formula. Because every process assumes the veracity of those assignments, each of them is called an assumption. This comes from the idea of a guiding path.

Guiding paths. Guiding paths are a concept introduced by PSATO (ZHANG; BONACINA; HSIANG, 1996) and provide a way to describe a path of decisions. As presented in (ZHANG; BONACINA; HSIANG, 1996), a guiding path is a list of decisions for variables (literals) and whether they are open (tested one way) or closed (tested both ways) (ZHANG; BONACINA; HSIANG, 1996), respectively represented as 1 and 0 (ZHANG; BONACINA; HSIANG, 1996). If a literal is open, it was only tested one way and the SAT Solver knows it needs to test the other polarity assignment, while if it is close, it does not need to tested it (ZHANG; BONACINA; HSIANG, 1996). PSATO uses the concept of guiding paths both for splitting the problem for many threads and for allowing a slave to record its work, thus enabling it to pick up where it left off in case it needs to stop, for any reason (ZHANG; BONACINA; HSIANG, 1996).

If we only need to parallelize the problem, without the need to pick up where we left off, we can simply store the literals describing assignments to variables, without the need to store whether the literal is open or closed. That is true because, if we distribute correctly the search space, including it entirely, we can simply consider every literal as closed, because other threads will search through the search space of the different polarities of these literals. The thread only needs to search the polarities attributed to it, since they do not need to have their polarities flipped, they are closed. Although different from PSATO, it is a possible approach.

In a guiding path, the selection of more variables allows the division of the problem in even more parts. In such case, a set of variables is chosen. Once this is done, there are 2^n guiding paths that define sub-formulas to be distributed. For instance, if the variables v_1 and v_2 are chosen, there are four guiding paths that provide four sub-formulas: (v_1, v_2), $(\neg v_1, v_2)$, $(v_1, \neg v_2)$ and $(\neg v_1, \neg v_2)$. Each sub-formula is the result of adding the assumptions to the initial formula and may be solved by a different processor. The guiding paths must always include every single possible combination of values for the chosen variables, otherwise, not all search space will be traversed.

As mentioned before, even though the distribution of one sub-formula per process may seem a good division of the problem, this generally fails as a workload balance (SINGER, 2006). Since not all the search space needs to be covered, due to conflict detecting, it is most likely that the search in one of those parts will be much greater than the others, because distribution of solutions is non-uniform in average (SINGER, 2006).

To mitigate this problem, some SAT Solvers use heuristics to choose the variables with which divide the space, such as VSIDS (HAMADI; JABBOUR; SAIS, 2009). Since VSIDS is an heuristic for making decisions (refer to Section 2.3.5) it is used to find variables to partition the search-space.

In (SINGER, 2006) portfolio is used along with divide-and-conquer with VSIDS. It uses this strategy called portfolio to attenuate the work balance problem. When a bad work balance is found, the strategy is switched to portfolio. The portfolio strategy is discussed later.

Another way to mitigate this problem is to use a dynamic distribution of sub-formulas. Instead of giving one sub-formula per process at their creations, the sub-formulas are dynamic distributed as the search progresses. As a process finishes its work, it receives more. There are many ways to dynamic distribute sub-formulas, Sections 2.4.2.1.2 and 2.4.2.1.3 present two of them.

2.4.2.1.2 Task Farm

Because statically giving one sub-formula to each processor is not a good workload distribution, strategies to dynamically (during execution) distributing the formulas are available. One of them is called by (GIL; FLORES; SILVEIRA, 2008) as Task Farm and consists of having a master process responsible for splitting the formula with guiding paths and sending to slaves (GIL; FLORES; SILVEIRA, 2008). There may be more than one sub-formula per slave (GIL; FLORES; SILVEIRA, 2008), but each receives one at a time. Once a slave is done with its share, it sends its results to the master and waits for more work (GIL; FLORES; SILVEIRA, 2008). A master sends more work to the slaves while no solution is found or while there are sub-formulas to be solved (GIL; FLORES; SILVEIRA, 2008). When a solution is found, the master terminates the execution and returns SAT. If all sub-formulas are UNSAT, the master terminates and returns UNSAT (GIL; FLORES; SILVEIRA, 2008).

This dynamic distribution of sub-formulas allows the SAT Solver to divide the problem in more parts than the number of processes. With that, it is possible to cope with the mentioned problem of workload balance. It is up to the developer to determine (or create a heuristic to determine) in how many parts the formula should be split, trading off communication for better workload balance.

2.4.2.1.3 Dynamic Work Stealing

Dynamic work stealing is another smart strategy to solve the problem of bad workload balance brought by static division of the problem. The strategy here is to distribute the problem among the process and then, when one process is done with its share, it can "steal" work from others that are not (JURKOWIAK; LI; UTARD, 2005).

One way to do that is, as presented in (JURKOWIAK; LI; UTARD, 2005), to have a master that only receives requests of work. When a slave is done with its share, it requests the master for more, which, in its turn, requests work to another slave and then sends it to the first slave. This way, no slave is idle while the problem is not solved (except when waiting the master's response) and it is only responsibility is to search for the solution. The downside is that workload management is centered in the master. If too many requests are received, it may create a bottleneck, where the slave spends too much time waiting for the master, in an idle state, rather than processing the problem.

This strategy requires the processes to the preemptive (JURKOWIAK; LI; UTARD, 2005). That is, a process must be able to receive at any given time a command to stop its work, split the remaining pending work, send one part and resume from the other.

It is common to have very small parts of the problem stolen, due to the lack of

balance of the search tree (CHU; STUCKEY; HARWOOD, 2008). This may be a big downside of this strategy, since it may cause an increase in the communication and, possibly, "The Ping-Pong Phenomenon" in some moments.

It also causes processing time loss, since the process that receives the request has to stop processing and split its problem (CHU; STUCKEY; HARWOOD, 2008). One possible solution to cope with this problem, used by PMiniSAT, is to have a centered queue which hold the subproblems and from which the problems are stolen (CHU; STUCKEY; HARWOOD, 2008). This removes the need of sending the sub-formulas (CHU; STUCKEY; HARWOOD, 2008).

2.4.2.2 Competitive Parallel SAT Solvers

The idea of a competitive SAT Solver became with the Portfolio approach (IRFAN; MANTHEY, 2013), which seems to be the only approach of competitive SAT Solvers. This approach, instead of dividing its search-space in a cooperative manner, attempts to compete, each thread solving the same formula (IRFAN; MANTHEY, 2013) in parallel. Each thread has a different strategy, with either different SAT Solvers (IRFAN; MANTHEY, 2013) or different configurations of SAT Solvers (IRFAN; MANTHEY, 2013), in parallel. Because different strategies have different effects in the solving process, one of these SAT Solvers is more likely to find the solution, so they compete for an answer and the gain is that the final solving time will be the one of the faster SAT Solver. It is important to point out that competitive SAT Solvers may also be cooperative, through learnt clause sharing (HAMADI; WINTERSTEIGER, 2013), which enhances the search process with little communication (HAMADI; WINTERSTEIGER, 2013).

2.4.2.2.1 Portfolio

Portfolio strategies take advantage of the fact that different strategies have different efficiencies on the search (XU et al., 2008). Therefore, many strategies, or algorithms, are run in parallel (XU et al., 2008), a portfolio of solvers, with the same input. Thus, the input is not divided, but run in parallel by different solvers. If one of the solvers fits well with the input, the problem should be efficiently solved. In (GIL; FLORES; SILVEIRA, 2008) the division of the problem by strategy rather than the input is defined as a functional partition.

A portfolio strategy sometimes also includes the notion of choosing the better algorithms (XU et al., 2008). Instead of running the instance with different arbitrary strategies, heuristics are used to choose them to better perform to the input (XU et al., 2008).

Since the portfolio contains solvers that solve the problem independently, it would

be possible to include parallel solvers. Thus, a parallel portfolio SAT Solver would divide the problem in different problems, each of which could further parallelize the problem. This second level of parallelization, would, then, divide the problem and solve it in parallel. Although possible, such idea was not found in this research.

It is important to point out that the use of portfolio does not exclude the possibility of clause sharing. ManySAT (HAMADI; JABBOUR; SAIS, 2009) uses both clause sharing and portfolio approach. Even using different strategies, the different process can take advantage of clauses learnt from other process. Since the process use different strategies, it is possible that one easily finds clauses that others would not find easily. In this scenario, clause sharing could make one process compensate limitations of the others, by sending clauses they could not easily learn.

2.4.3 Shared Memory or Distributed Memory

When solving a problem in parallel, processors must be able to communicate with each other (QUAMMEN, 2005). There are two basic kinds of communication processors use: a shared-memory approach, using a single address space (QUAMMEN, 2005); or a distributed-memory one, in which the components are connected by a communication network (QUAMMEN, 2005) through which they communicate (QUAMMEN, 2005) by passing message (LEWIS; SCHUBERT; BECKER, 2007). The first step before the development of a parallel SAT Solver is to choose which of this paradigms will be used.

The usage of one or another of these communication paradigms implicates on how the components are programmed (QUAMMEN, 2005). There are SAT solvers built for either paradigms (SINGER, 2006), and the choice of the communication requires different characteristics from the SAT Solvers.

Generally, the usage of shared-memory parallelism results in less communication overhead, because message passing is not used and memory accessing is much faster than common network communication. On the other hand, distributed-memory allows the software to avail of computer clusters, which provide much greater computational power than multiprocessors. The choice between one or another generally lies on the need and availability of computation power.

2.4.4 Clause Sharing

Most of known SAT Solvers use similar CDCL approaches (HAMADI; JABBOUR; SAIS, 2009), including parallel ones. It brings the possibility of clause sharing. Every time one processing unit of the SAT Solvers learns a clause, this clause can be used to shorten the search of every other unit, if they also learn it.

Clause learning is an important characteristic of current SAT Solvers, that greatly shortens the search space. On the other hand, in parallel solvers, if not implemented with a smart strategy, sharing may cause an unwanted increase in communication. One solution, used by ManySAT, is to only share learnt clauses when they don't exceed a predefined limit size (HAMADI; JABBOUR; SAIS, 2009). In ManySAT, this limit is 8 literals (HAMADI; JABBOUR; SAIS, 2009).

Another problem in clause sharing is the strategy used when receiving a clause. Differently from learning clauses in sequential SAT Solvers, receiving a clause requires a strategy to analyze and react to the new clause, since it was unknown so far, and was not generated in the current search branch. The following scenarios are described by ManySAT as what may occur when receiving learnt clauses (HAMADI; JABBOUR; SAIS, 2009), but are generic enough to be applied in other parallel SAT Solvers:

1. The new clause may be conflicting in the current context. In this case, the conflict must be identified to allow a backtracking to a previous level.
2. It may be unit in the current context, depending on the decision and propagated literals. In this case, it may cause a unit propagation to occur.
3. It may be satisfied in the current context. ManySAT watches such clauses to take advantage in the future, but it does not have a real instant importance.
4. The new clause is not falsified, satisfied or unit. In this case, ManySAT watches the clause. This clause may cause conflicts or implications in the future.

Some SAT Solvers, such as MiraXT (SCHUBERT et al., 2007), use a set of clauses shared among threads. This approach tends to decrease the communication between threads, since it is not necessary to send clauses, but it generally requires the usage of locks, when modifying the set of clauses, so a trade-off between communication and shared critical zones is necessary. Also, this approach is not possible in distributed memory.

2.5 CUDA

Parallel computing has increased its importance it the past years. As (SANDERS; KANDROT, 2010) puts it: "[...] nearly any aspiring programmer needs training in parallel programming to be fully effective in computer science". Over the years, parallel computing has become present in most devices (SANDERS; KANDROT, 2010), becoming the new way to improve efficiency (SANDERS; KANDROT, 2010), and carrying with it this need of knowledge in parallel programming.

GPUs environments were initially developed for processing graphical operations. Applications employing 3D graphics had an increasing demand by the mid-1990s (SANDERS; KANDROT, 2010), which caused significant improvement and ignited the production of affordable graphic accelerators (SANDERS; KANDROT, 2010). In this context, ways to actually let the programmer have control over the computation in GPUs have arrived, rather than only interacting with APIs, such as OpenGL or DirectX, and CUDA was one (SANDERS; KANDROT, 2010). CUDA was created by NVidia as a way to allow general programming on GPU, or GP-GPU (SANDERS; KANDROT, 2010).

CUDA-enabled processors have an ALU intended for GP-GPU (SANDERS; KANDROT, 2010), they comply with IEEE requirements for single-precision floating-point (SANDERS; KANDROT, 2010) and have a set of instructions tailored for general computing, instead of having one created for graphics processing (SANDERS; KANDROT, 2010). These and other features made CUDA a much better environment for GP-GPU. Since its creation, many applications have taken advantage of CUDA (SANDERS; KANDROT, 2010), reaching orders or magnitude in performance improvements (SANDERS; KANDROT, 2010).

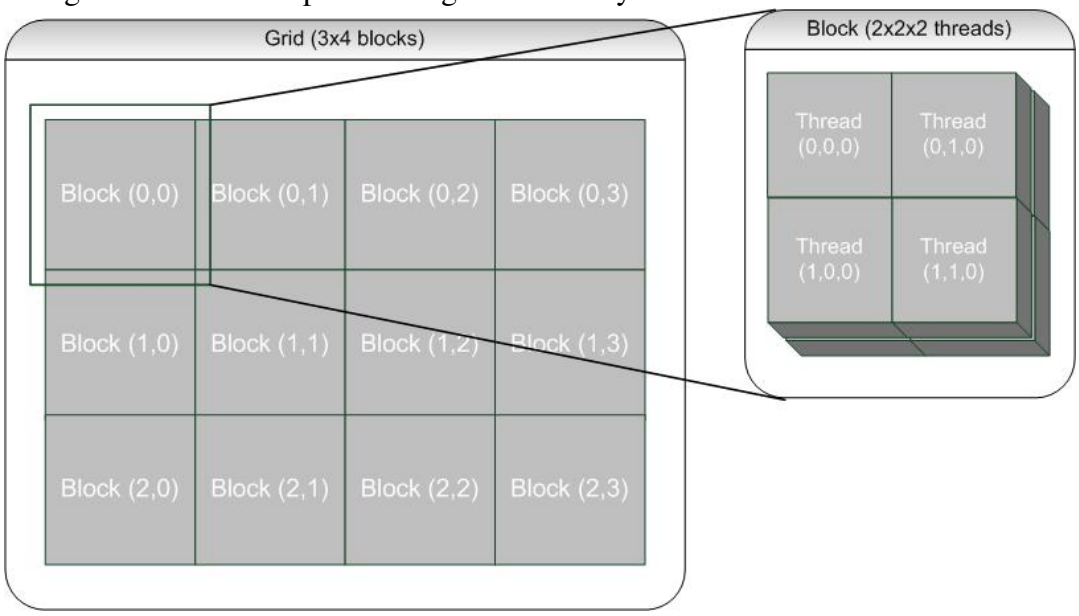
CUDA C is a programming language that allows both development of code to be run on the CPU (the host) and the GPU (the device) (SANDERS; KANDROT, 2010). This allows the development of CPU code that directly interacts with GPU code, by a simple routine call (SANDERS; KANDROT, 2010). CUDA uses blocks and threads to parallelize code.

2.5.1 Blocks and Threads per Block

When we need to parallelize something in CUDA, we use blocks and threads to accomplish that. Every block has a set of threads (SANDERS; KANDROT, 2010) and each different thread in the same block runs the same code. Different blocks and threads receive different indices (SANDERS; KANDROT, 2010), that allows them to behave differently. An example of this structure is depicted in Figure 2.5, with a grid with $3x4$ blocks and each block with $2x2$ threads.

Blocks are organized in a two-dimensional group, called a grid, and threads in three-dimensional groups (SANDERS; KANDROT, 2010) within each block. This way, we identify any block by a ordered pair as its index (x and y coordinates) and a thread in any block by a ordered triple as it index (x, y and z coordinates). The index of a thread is only different from the indices of threads in the same block, so to identify uniquely a thread, its block index must be used along with the index of the thread. This multi-dimensional scheme is used to make is simpler for solving problems with matrices (SANDERS; KANDROT, 2010) and other structures with more dimensions. For example,

Figure 2.5: An example showing the hierarchy of blocks and threads in CUDA.



if we need to compute over a NxN matrix, we can simply create a block with $NxNx1$ threads and each processing the matrix position of their x and y indices.

Threads within a same block are able to access the same shared memory structures (SANDERS; KANDROT, 2010). That is, there are some structures created in a shared memory location (refer to section 2.5.2), that can be accessed by many threads, all threads in the same block.

2.5.1.1 SIMT Model

In CUDA, threads are organized in groups of 32 threads, called *warps* (CUDA C Programming Guide, 2013). Threads in different warps start execution at the same instruction (CUDA C Programming Guide, 2013), but have different instruction address counters, so they may execute differently, branching in different directions (CUDA C Programming Guide, 2013).

On the other hand, threads in the same warp execute a common instruction at a time (CUDA C Programming Guide, 2013), with all threads carrying out the same instructions. If by a data dependency, one or more threads in the warp need to execute instructions that others do not, given conditional operations, these threads execute this path, while the other threads become idle until they all have the same instructions to execute again (CUDA C Programming Guide, 2013). This behavior only happens within a warp and threads in different warps may execute different operations in parallel (CUDA C Programming Guide, 2013). Full efficiency is reached when all threads in a warp are executing the same instruction (CUDA C Programming Guide, 2013), with no idle threads waiting for instructions they can execute.

This behavior happens because CUDA was built to be an SIMD architecture, capable of running the same instructions in different data. In this scenario, each thread is responsible for a part of the data, running the same instructions on it, behavior called SIMT. However, as an SIMT, it is capable of branching differently, in different threads. According to (CUDA C Programming Guide, 2013), the programmer can ignore this SIMT behavior, with the cost of efficiency.

All threads in a warp come from the same block (CUDA C Programming Guide, 2013) and are organized in warps of 32 threads (CUDA C Programming Guide, 2013). If a block has a number of threads that is not multiple of 32, there will be inactive threads in some warp (CUDA C Programming Guide, 2013), because a warp is always created with 32 threads. For example, a block with 34 threads will result in a second warp with 2 active and 30 idle threads. For that reason, (Cuda C Best Practices Guide, 2013) advises the use of a number of threads multiple of 32.

2.5.2 Memory Structure

CUDA provides many memory spaces for the threads (CUDA C Programming Guide, 2013), the choice of where to have data structures affect the efficiency of the software. Here are the basic memory spaces that CUDA provides:

Local Memory : Every thread has its local memory (CUDA C Programming Guide, 2013). This memory is only accessible by the thread, without sharing.

Shared Memory : Shared structures, created in shared memory, are created to be accessed by all threads of a block (SANDERS; KANDROT, 2010). A copy of each structure (basic type, vector, etc.) is created for each block, and every thread in the block can access it. These structures are created in a memory that resides in the GPU, rather than an extern DRAM memory, which makes it much faster (SANDERS; KANDROT, 2010).

Global Memory : Global memory is the memory space that allows all threads to access their content. Any data stored in this space can be accessed by any thread in any block (CUDA C Programming Guide, 2013), as opposed to shared memory, which only allows threads in the same block to access their content.

Constant Memory This memory space is used to store unchangeable data (SANDERS; KANDROT, 2010). Data stored in this memory is treated differently from other memory spaces (SANDERS; KANDROT, 2010). It is generally faster, because: (I) it allows broadcast of data to 'nearby' threads (within the same half-warp with 16 threads) (SANDERS; KANDROT, 2010), generating less memory traffic, for many reads, about only 6% of original traffic (SANDERS; KANDROT, 2010) and (II) it is cached. Because of these two reasons, this may be a good option, if the cost of having an unchangeable and size

fixed memory is acceptable, depending on the case.

Texture Memory : Texture memory is a memory provided by CUDA specially made for some graphics (CUDA C Programming Guide, 2013), though it may still be effective for other GPU computing applications (SANDERS; KANDROT, 2010). This is another read-only memory (SANDERS; KANDROT, 2010) and it is also cached (SANDERS; KANDROT, 2010), as constant memory. This memory space is designed to accelerate software that has spatial locality (SANDERS; KANDROT, 2010). In a two-dimensional structures, nearby memory that is store in different lines in the matrix, but similar column would not be cached together in regular CPU memory, but texture memory is made for this scenario (SANDERS; KANDROT, 2010) and provides improvement in these cases (SANDERS; KANDROT, 2010).

We can see that CUDA provides many different memory spaces. Because CUDA has many arithmetic units, generally the bottleneck is not the arithmetic throughput (SANDERS; KANDROT, 2010), but the memory bandwidth (SANDERS; KANDROT, 2010). In many cases, it is hard to keep feeding these units with data to compute (SANDERS; KANDROT, 2010). An intelligent use of these spaces may bring big gains in efficiency.

2.5.3 Atomic Operations

Because there is parallel computation in CUDA, sometimes it is necessary to guarantee that some operations over shared structures are carried out atomically, that is, only one thread at a time. Atomic operations are operations provided by CUDA that are always entirely executed by one thread before they can be executed by another (SANDERS; KANDROT, 2010), that is, atomically. CUDA provides many atomic operations (SANDERS; KANDROT, 2010). Therefore, these atomic operations are ways to synchronize threads working in a cooperative manner, making sure data will not be corrupted during the execution.

Among the operations we have: the sum of two numbers, carried out atomically (CUDA C Programming Guide, 2013), as well as subtraction; increment and decrement of a number; exchange of two values of variables; min and max, allowing the update of a variable if its previous value is greater/smaller than the one we want to set; Boolean operations (CUDA C Programming Guide, 2013); among others. These operations generally return the previous value the changed variable carried (CUDA C Programming Guide, 2013). That is very useful, because some applications may need to used and set a value atomically, which would be impossible if those two operations were carried out as two operations.

3 RELATED WORK

We present here some related work concerning RePaSAT and SAT-PaDdlinG. In Section 3.1 we present some parallel CPU SAT Solvers, mostly applying the parallelization techniques presented. In Section 3.2, we present some GPU SAT Solvers, with different characteristics from those presented in Section 3.1. At last, in Section 3.3 we present some automatic parallelization tools, like RePaSAT.

3.1 Parallel CPU SAT Solvers

This section presents a selection of parallel SAT Solvers. The criteria for this selections was mainly frequency in literature and references. There are other parallel SAT Solver found that are not presented here. However, the present ones cover a reasonable number or characteristics and uses.

3.1.1 ManySAT

ManySAT (HAMADI; JABBOUR; SAIS, 2009) is a shared-memory SAT Solver (HAMADI; JABBOUR; SAIS, 2009) that uses a portfolio approach (HAMADI; JABBOUR; SAIS, 2009). ManySAT creates a portfolio of 4 solver instances that are run in parallel (HAMADI; JABBOUR; SAIS, 2009). Each instance varies in solving strategies, such as restarts and polarity choice algorithms (HAMADI; JABBOUR; SAIS, 2009). ManySAT was empirically shown to work well both on SAT and UNSAT instances of problems (HAMADI; JABBOUR; SAIS, 2009).

ManySAT is frequently mentioned in literature and (HAMADI; JABBOUR; SAIS, 2009) presents very good experimental results for ManySAT. A drawback is the fact that ManySAT only allows the run of 4 instances in parallel (HAMADI; JABBOUR; SAIS, 2009). Since the input of the problem is not divided, but run with different strategies, adding new instances would require new strategies to be implemented. For real time gain, such strategies would have to perform better than the other four for some classes of

problems.

For clause sharing, each process has their own set (HAMADI; JABBOUR; SAIS, 2009). The clauses are shared among them, as long as its size does not exceed a limit, set as 8 literals (HAMADI; JABBOUR; SAIS, 2009). The reaction when a process receives a clause are described in Section 2.4.4.

3.1.2 PMiniSAT

PMiniSAT (CHU; STUCKEY; HARWOOD, 2008) is a parallelization of a well-known sequential SAT Solver called MiniSAT (CHU; STUCKEY; HARWOOD, 2008). It uses dynamic work stealing to divide the problem (CHU; STUCKEY; HARWOOD, 2008). The work stealing is carry out in the longest running thread and from as high in the search tree as possible (CHU; STUCKEY; HARWOOD, 2008). As mentioned before, PMiniSAT uses a centered queue from which the subproblems are stolen (CHU; STUCKEY; HARWOOD, 2008). It avoids the overhead processes would have if they sent part of the problems to others and also avoids the waiting for response (CHU; STUCKEY; HARWOOD, 2008).

For clause sharing, it uses two schemes (CHU; STUCKEY; HARWOOD, 2008). One is, like ManySAT, the sharing of clauses between threads as long as its size is not greater than a threshold. In PMiniSAT, this threshold is 5 literals (CHU; STUCKEY; HARWOOD, 2008).

The second scheme is complementary to the first (CHU; STUCKEY; HARWOOD, 2008). Clauses, which may have size greater than 5, are shared among threads with similar guiding path (CHU; STUCKEY; HARWOOD, 2008), the decisions made up to the moment. There is, though, a threshold for the effective size of the clause relative to the receiver thread (CHU; STUCKEY; HARWOOD, 2008). The effective size is the size after removing the literals that are interpreted as false by the decisions in the receiving thread. A learnt clause from a thread should be more important to a thread whose decisions are similar to the thread that generated it (CHU; STUCKEY; HARWOOD, 2008).

3.1.3 MiraXT

MiraXT is a reimplementation of Mira (SCHUBERT et al., 2007). MiraXT is multi-threaded SAT Solver that uses the shared-memory paradigm (SCHUBERT et al., 2007). One important characteristics is that each clause is only stored once in memory, which is shared among the threads, and locks are used for insertion (SCHUBERT et al., 2007). MiraXT uses conflict-driven clause learning.

Because, once inserted, the clause is read-only, the sharing itself can be done

without locks (SCHUBERT et al., 2007). That is, more than one thread can access the same clause at the same time. Locks are used, though, when inserting or removing clauses (SCHUBERT et al., 2007). MiraXT uses an algorithm to determine the inactive clauses to be removed (SCHUBERT et al., 2007).

To workload distribution, it uses a dynamic Divide-and-Conquer strategy, similar to Dynamic Work Stealing. A thread starts the process and check if there are threads needing work (LEWIS; SCHUBERT; BECKER, 2007). If so, it splits its formula (the initial after preprocessing) and stores half in a shared queue (LEWIS; SCHUBERT; BECKER, 2007). Other threads take work and do the same, storing part of their work, when there are threads needing work (LEWIS; SCHUBERT; BECKER, 2007). When a thread returns UNSAT, it requests more work (LEWIS; SCHUBERT; BECKER, 2007). While there are running and idle threads, the running ones stop and split their sub-formula, putting in the queue (LEWIS; SCHUBERT; BECKER, 2007). This is done until all threads are idle, which means the formula is UNSAT, or one finds a solution (LEWIS; SCHUBERT; BECKER, 2007).

3.1.4 PMSat

PMSat (GIL; FLORES; SILVEIRA, 2008) is a parallel SAT Solver that uses a master-slave hierarchy to divide the workload, by using Task Farm (GIL; FLORES; SILVEIRA, 2008). The division of the formula is configurable and may use different strategies (GIL; FLORES; SILVEIRA, 2008). PMSat uses two strategies, either choosing the most frequent variables or the variables that occur in bigger clauses (GIL; FLORES; SILVEIRA, 2008).

PMSat uses MPI, which is a message passing interface, and, therefore, is aimed at distributed-memory paradigm (GIL; FLORES; SILVEIRA, 2008). PMSat is built over MiniSAT, thus the slaves works as MiniSAT instances that solve sub-formulas.

For clause sharing, the master centralizes this communication. When a slave sends the result for its sub-formula, it also sends its learnt clauses (GIL; FLORES; SILVEIRA, 2008). The master receives and stores them (GIL; FLORES; SILVEIRA, 2008). When sending more work to other slaves, the learnt clauses are also sent (GIL; FLORES; SILVEIRA, 2008). This mechanism is optional (GIL; FLORES; SILVEIRA, 2008). It is possible to limit the number and size of shared clauses (GIL; FLORES; SILVEIRA, 2008).

It is not made clear if the centralization of the master causes too much overhead. PMSat implements a master that is responsible for sending work and receiving results, receiving shared clauses (although optional), creating assumption with two possible algorithms to send to slaves, eliminating assignments that conflict with received clauses, among others (GIL; FLORES; SILVEIRA, 2008). Clause sharing results in storing every

learnt clause, from every slave (although old are removed, when threshold is reached), which requires the master to keep and manage which slaves have received and which clauses, not to send them again.

This overhead could, possibly, drastically reduce the efficiency of the master. Since the slaves are dependable of the master to receive more work, they may become idle, waiting for a busy master in such a scenario. It is not mentioned in (GIL; FLORES; SILVEIRA, 2008) if this problem was really observed.

3.1.5 PaSAT

PaSAT (SINZ; BLOCHINGER; KüCHLIN, 2001) is a parallel SAT Solver developed to be used both on a shared-memory or distributed-memory architecture (SINZ; BLOCHINGER; KüCHLIN, 2001). It uses the Distributed Object-Oriented Threads System (DOTS), a toolkit for C++ for parallel programming (SINZ; BLOCHINGER; KüCHLIN, 2001). This toolkit allows the software to be used in a distributed-memory environment (SINZ; BLOCHINGER; KüCHLIN, 2001). It uses a dynamic Divide-and-Conquer space, similar to Task Farm, to divide the problem, through guiding paths (SINZ; BLOCHINGER; KüCHLIN, 2001). It also implements clause sharing, by storing them in a data structure shared among threads (SINZ; BLOCHINGER; KüCHLIN, 2001). Because a shared structure is used, threads can only access the clauses generated by threads running in the same machine (SINZ; BLOCHINGER; KüCHLIN, 2001), which share the same memory space.

3.1.6 SArTagnan

SArTagnan (KOTTLER; KAUFMANN, 2011) is a portfolio SAT Solver that promises to improve efficiency for not using locks in clause sharing (KOTTLER; KAUFMANN, 2011). It is developed to a shared-memory architecture by using threads (KOTTLER; KAUFMANN, 2011). It shares clauses among threads (KOTTLER; KAUFMANN, 2011). The clauses are only stored once in memory and threads have references to them (KOTTLER; KAUFMANN, 2011). As each clause is an immutable structure, it may be access without locks, but masks are used to provide mutual exclusion of some operations.

The biggest promise brought by SArTagnan is the clause sharing without locks. That is the main different between SArTagnan and ManySAT. They are both portfolio SAT Solvers, but it is possible that ManySAT has a greater communication overhead, due to clause sharing.

3.1.7 GridSAT

GridSAT is a parallel SAT Solver, based on the sequential SAT Solver zChaff, developed to Computational Grid execution (CHRABAKH; WOLSKI, 2003). Thus, it uses a distributed-memory paradigm. It was built to allow the use of large sets of heterogeneous computational resources (CHRABAKH; WOLSKI, 2003). By the time (CHRABAKH; WOLSKI, 2003) was published, it was the world-record holder among all SAT Solvers tested at SAT2002 Solver competition (CHRABAKH; WOLSKI, 2003). It was developed using the EveryWare development toolkit (CHRABAKH; WOLSKI, 2003).

It implements clause share, in which each "client" (which is, in fact, a slave) may send and receive clauses (CHRABAKH; WOLSKI, 2003). Every client has their own clauses and include the received ones (CHRABAKH; WOLSKI, 2003). Only small clauses are shared and the maximum size of a share clause is a parameter (CHRABAKH; WOLSKI, 2003).

GridSAT has a master process, which manages the resources, manages the clients and distributes work (CHRABAKH; WOLSKI, 2003). The distribution of work is done differently. The execution starts with one client and, when the client deems it is running out of memory or has a subproblem that is too big, it contacts the master and requests to split the work (CHRABAKH; WOLSKI, 2003). The master allocates a new client and informs the first client (CHRABAKH; WOLSKI, 2003). The first client splits its formula and send to the other client, along with some learnt clauses (CHRABAKH; WOLSKI, 2003). This is similar to dynamic work stealing, except the work is not stolen, but given. Because the execution starts with one client and this number in increase, there is part of the execution that is done with less processes than necessary, which may affect performance, though this is not mentioned.

3.1.8 Comparison Between SAT Solvers

This sections aims at presenting a comparison between the SAT Solvers presented previously. We present here a comparison between their basic characteristics, specially concerning architecture paradigm, workload distribution and clause sharing. The Table 3.1 illustrates this characteristics for each SAT Solver.

Choosing between those SAT Solvers for a specific class of problems seems to be a hard task. It is not present in literature, in general, the main class of problems each SAT Solver is better at. That is probably because, in the end, independently were the formula comes from, they are all in the same format, and is hard to distinguish a class. There is, nevertheless, a difference between random formulas, industrial, and handmade ones, sometimes mention in literature and used as classes of problems in the SAT Competition

Table 3.1: Main characteristics of SAT Solvers

SAT Solver	**Architecture Paradigm**	**Workload Distribution**	**Clause Sharing**	**Main Limitations**
ManySAT	Shared-memory	Portfolio	Stored separately, shared through lockless queues, max size to share of 8 literals	Allows only 4 threads
PMiniSAT	Shared-memory	Dynamic Work Stealing (with centered work queue)	Stored separately, two schemes: share small and relatively small with similar processors	
MiraXT	Shared-memory	Dynamic Divide-and-Conquer (Work Stealing like)	Clauses are stored once and shared, locks used for insertion	Centered shared work queue, threads must stop to split work
PMSAT	Distributed-memory	Task Farm	Stored separately, sent from slave to master, then sent to slave again	Many tasks centralized in the master
PaSAT	Shared and distributed-memory	Dynamic Divide-and-Conquer (Task Farm like)	Clauses stored once and shared, only with threads in the same machine	It is not possible to share clauses with threads in other machines
SArTagnan	Shared-memory	Portfolio	Clauses stored once and shared, masks are used for coherency	
GridSAT	Distributed-memory	Dynamic Divide-and-Conquer (Work Stealing like)	Stored separately, send from client to client	Number of clients is increased during process. Few clients in the beginning.

(The International SAT Competitions Web Page, 2013).

There is also SAT and UNSAT problems, but this distinction is generally used in benchmarks, since we usually do not know whether an instance is SAT or not. If this characteristic is know, it would be possible to take advantage of it. ManySAT is one SAT Solver that seems to work for both SAT and UNSAT problems (HAMADI; JABBOUR; SAIS, 2009). Other classes of problems were not found during this research.

Empirical evaluation of SAT Solver seems to be a way to choose among them. One place to find evaluations of many SAT Solvers, separated in the mention categories is the SAT Competitions (The International SAT Competitions Web Page, 2013).

3.1.9 Overall Analysis on CPU SAT Solvers

All presented SAT Solvers in this Section provide parallelism and many of them are widely use. However, there is a characteristic we are looking for that most of them do not show: massive parallelism. Most of these SAT Solvers are limited due to the environment they are meant to run in: a CPU. Current CPUs alone are not able to provide massive parallelism and because RePaSAT imposes an overhead for translating the code into an NP-Complete problem, these SAT Solvers lack an important characteristic we seek.

GridSAT provides massive parallelism. However, it requires a cluster of computers, an expensive and hard to find environment, which, in this sense, cannot be better than our other alternative: the GPU environment. This is a cheap and easy to find environment that may give us the parallelism we look for. For that reason, in Section 3.2 we present some SAT Solvers specially developed to run in a GPU environment.

3.2 GPU SAT Solvers

We have studied some SAT Solvers for GPU. Like SAT-PaDdlinG, the GPU SAT solver we present in Chapter 4, they generally take advantage from the parallelism provided by a GPU, but none of them fully parallelizes a DPLL procedure on GPU. One characteristic common for most of the SAT Solvers found is that they usually solve the 3-SAT problem. Though the 3-SAT problem is as expressive as the SAT problem, since any SAT formula can be expressed as a 3-SAT formula, reducing a formula to 3-SAT causes an increase in size, including the number of variables, which could be avoided if the clause did not have a size limit.

3.2.1 GPU4SAT

GPU4SAT (DELEAU; CHRISTOPHE; KRAJECKI, 2008) is a SAT Solver that runs on GPU using CUDA (DELEAU; CHRISTOPHE; KRAJECKI, 2008). GPU4SAT is an incomplete approach (not necessary finds the answer) that uses a matrix representation for the clauses and matrix multiplications to verify interpretations (DELEAU; CHRISTOPHE; KRAJECKI, 2008). It uses the number of satisfied clauses to determined which literal to flip (DELEAU; CHRISTOPHE; KRAJECKI, 2008). Different interpretations are distributed among the threads and those are improved as described in an attempt to find a solution (DELEAU; CHRISTOPHE; KRAJECKI, 2008).

3.2.2 BCP Parallelization

This SAT Solver does not have a name. It was proposed by (FUJII; FUJIMOTO, 2012) and is a parallelization of the BCP for 3-SAT instances, by using a 3SAT-DC (3SAT divide-and-conquer). As mentioned in Section 2.2.2, BCP is the procedure in DPLL SAT Solvers that determines whether an interpretation causes a conflict as well as generates implications from this interpretation. They use a divide-and-conquer approach to search for the solution, while parallelizing every call of BCP (FUJII; FUJIMOTO, 2012). Its parallelizing approach is neither portfolio nor search-space split for it does not parallelize the input, only the BCP procedure.

This approach is the closest to a DPLL SAT Solver on GPU we could find, with a few differences. Firstly, it does not entirely parallelize the problem. When executing, whenever the BCP procedure is called, it can be automatically parallelized using the algorithm presented. The second difference is that it does not parallelize through the input or using a competitive approach. This is a different approach and though it is aimed at a DPLL SAT Solver, different from many common parallel SAT Solvers, it does not parallelize the solving process as a whole.

3.2.3 SAT Solver for Random Small Instances

This is another approach without a name, presented by (MEYER; SCHONFELD; SCHONFELD, 2010). It is another 3-SAT approach to CUDA. Since all its clauses are 3 literals long, it chooses a clause and for its literals tests three combinations of variables attributions: first literal true; first literal false and second true and first and second literal false and third true (MEYER; SCHONFELD; SCHONFELD, 2010). It is focused in random instances (MEYER; SCHONFELD; SCHONFELD, 2010), which are generally hard to solve, even when small, for lacking internal structures to be exploited (MEYER; SCHONFELD; SCHONFELD, 2010). Since it predicates on small random instances (MEYER; SCHONFELD; SCHONFELD, 2010), this approach is inappropriate for large

instances.

This approach predicates on the fact that random instances are generally small and hard to solve, even with optimizations they don't use (MEYER; SCHONFELD; SCHONFELD, 2010). Random instances are harder because, since they are not generated from a real problem, they lack symmetries and other structures that SAT Solvers exploit to find the solution faster. Generally, SAT Solvers are used to solve real problems, but random instances usually have the purpose of testing the SAT Solvers. The real contribution brought by this technique is not clear.

3.2.4 Genetic Algorithm for SAT Solving

It is presented in (FEIER; LEMNARU; POTOLEA, 2011) genetic algorithms that solve both the SAT and the knapsack problem. As a genetic algorithm, it starts with a attribution and attempts to improve it (FEIER; LEMNARU; POTOLEA, 2011). Thus, it is an incomplete approach (FEIER; LEMNARU; POTOLEA, 2011) and may not find the answer. This approach seems to a fitting application for a GPU environment, since it works with functions over vectors. However, it is an incomplete approach, which is some cases may be a problem, if an answer is required. Furthermore, it does not have with it all the optimizations DPLL have gained over the years.

3.2.5 OpenCL-SAT

OpenCL-SAT is a hybrid SAT Solver proposed by (BECKERS et al., 2012). As a hybrid SAT Solver, it uses both complete and incomplete SAT Solvers to solve the problem (BECKERS et al., 2012). This SAT Solver runs an instance of a modified MiniSAT on the CPU and a parallel incomplete SAT Solver on GPU. The SAT Solver on the GPU improves the search on the CPU, as it improves its own solution. The VSIDS algorithm used by MiniSAT, which chooses the variables at each decision level, was replaced by the a verification in the GPUs SAT Solver evaluation for the variables.

This approach has a very big problems, it has to synchronize the CPU and the GPUs execution. Since the CPU chooses its next variables based on the SAT Solver on the GPU, it has to wait for it and it is essential for efficiency that both take approximately the same amount of time for each step.

The results of experiments presented on (BECKERS et al., 2012) are not at all optimistic. Not only (BECKERS et al., 2012) hardly presents any results, only averages of their results, the results are very bad. The time their parallel SAT Solver took to solve problems (in average) is longer than the time the sequential MiniSAT with VSIDS and it had to make more decisions, showing that this approach did not improve the decision making of variables, which was their point to start with. The authors just concluded that

their SAT Solver is too basic to compete with state-of-the-art SAT Solvers and is a step to a SAT Solver with potential.

3.2.6 Overall Analysis on GPU SAT Solvers

As we mentioned in Section 3.1.9, we expect to be able to solve CNF formulas on a GPU, to both take advantage of massive parallelism, without the extra cost of a cluster environment. Since GPUs are common and found in most devices, they provide the characteristic we look for. For that reason, we presented GPU SAT Solvers.

Although all presented SAT Solvers were developed to a GPU environment, there are some characteristic that stopped us from using them. A technical problem is that we have only been able to get the source code and executable of two of those SAT Solvers, the ones presented in Sections 3.2.3 and 3.2.4, even though the sources were requested to the authors of all SAT Solvers presented here.

Of those two SAT Solvers, the first one, presented by (MEYER; SCHONFELD; SCHONFELD, 2010) is a SAT Solver specially developed for small random instances. It takes advantage of the fact that random instances are generally reduced to brute force, even in optimized SAT Solvers, since they do not have a well established structure to take advantage of and proposes the idea of only solving small random instances on GPU. This SAT Solver does not fit our needs, since we do not generate random instances in RePaSAT, on contrary, they come from code and we expect them to have a structure and be optimized. Furthermore, since we want to massive parallelize the code, which was our objective when bringing the problem to a GPU environment in the first place, we expect the formulas to be large. A SAT Solver created for small instances, without optimizations to improve the search is inadequate.

The other SAT Solver, presented by (FEIER; LEMNARU; POTOLEA, 2011), is a genetic approach aimed, actually, at solving two NP-Complete problems on GPU: the SAT Problem and the Knapsack Problem. As a genetic algorithm, it is incomplete, that is, it may get stuck in a local maximum and the SAT Solver will have no way to determine whether the solution does not exist or could not be found. This is an important limitation, since we do not want our automatic parallelization to be able to run the software sometimes, even if it is most of the times. Because of that, we also discarded this as an option.

As for the others, even if we could get their code to use, they do not use the approach we expected to use, a DPLL SAT Solver. We aim at this approach, because it has been successful so far to solve very large formulas, with hundreds of thousands of variables, a characteristic seen in generated formulas by RePaSAT. These approach has been improved over the years and such improvements are necessary if we want to

parallelize big parts of the code. Of all presented SAT Solvers, only the one presented in Section 3.2.2 actually implements a parallelization of part of a DPLL SAT Solver, but it does not parallelize the entire code and its code was impossible to retrieve. Furthermore, it solves, along with some other approaches, 3-SAT formulas, different from the ones generated by Ursa, the translating tool we used, and translation to this format causes new variables, which increase the search space and make the problem even more difficult to solve.

3.3 Automatic Parallelization Tools

We present in this section other tools used for automatic parallelization. Automatic parallelization has already been approached in literature (AMBRUS, 2003; CORDES; MARWEDEL; MALLIK, 2010; PADBERG; MIROLD, 2012; BRADEL; ABDELRAHMAN, 2007). The Clairvoyance framework is presented in (AMBRUS, 2003), a framework for parallelizing sequential Java code. Clairvoyance does not automatically parallelize the code, but helps the developer to parallelize sequential code and distribute the problem (AMBRUS, 2003). It helps by analyzing the code and data structures, so the developer can manually parallelize the code (AMBRUS, 2003).

In (CORDES; MARWEDEL; MALLIK, 2010), it is presented a strategy for automatically parallelizing code aimed at embedded systems, by using integer linear programming to identify parallelism. Since it is targeted at embedded systems, it considers constraints relevant to embedded systems (CORDES; MARWEDEL; MALLIK, 2010).

ALCHEMY (PADBERG; MIROLD, 2012) is a tool to automatically parallelize R code. ALCHEMY translates R code into the Analysis Intermediate Representation (AIR) to parallelize it (PADBERG; MIROLD, 2012). Since R is a script language, it is parallelize on the fly (PADBERG; MIROLD, 2012), that is, as it is executed.

It is proposed in (BRADEL; ABDELRAHMAN, 2007) a strategy of using traces to automatically parallelize Java code. This strategy uses Java bytecodes to find traces and to run them in different threads.

3.3.1 Overall Analysis on the Automatic Parallelization Tools

There are some interesting approaches to automatic parallelism, but they all different from RePaSAT in one key aspect: they attempt to find parallelism in the code, rather that translating it to a parallelizable entity. Such characteristic may bring some very important gains. Since the SAT formula, the parallelizable entity in this case, is fully parallelizable, we are able to fully parallelize any part of the code we translate to a formula. Different from looking for iterations in a loop that can be run together or traces to

find the parallelism.

We believe that this new approach contrasts with the others and must be studied. Furthermore, RePaSAT is meant to parallelize imperative code, not restricted by environment, such as the one presented by (CORDES; MARWEDEL; MALLIK, 2010), aimed at embedded systems. Also, RePaSAT is fully automatic, not an auxiliary tool, as Clairvoyance. To all this, we also add the possibility of expressing declarative code, which helps programming itself.

4 SAT-PADDLING

Because RePaSAT, presented in Chapter 5, is supposed to bring parallelism advantage to software and it brings along overhead for this parallelization, we believe that it should be massively parallelized. However, most massively parallel environments are expensive and not easy to find. There is, nevertheless, the GPU environment, an environment capable of providing massive parallelism, while still being cheap and easy to find. For that reason, we started studied some GPU SAT Solvers, but we did not find one satisfying ours needs.

Many CPU SAT Solvers implement the DPLL procedure, due to its performance and optimizations. A GPU environment, such as CUDA, may not be completely suitable for the DPLL algorithm. CUDA uses an SIMD model (MEYER; SCHONFELD; SCHONFELD, 2010), and requires threads in the same warp to execute the same instructions (MEYER; SCHONFELD; SCHONFELD, 2010), unsuitable for the unpredictability of DPLL behavior, which is a highly control-flow algorithm. In fact, (MCDONALD, 2009) states that backtracking searches are impractical to parallelize on GPU, because it follows this model of parallelization. However, to the best of the authors' knowledge, the behavior of a parallel DPLL SAT Solver on a GPU environment is hitherto unknown, for we have not found any fully DPLL GPU SAT Solver. We deem necessary to study this algorithm empirically in this environment before entirely discarding it.

In this chapter, we present the description of the implementation of our GPU SAT Solver. SAT-PaDdlinG is a DPLL SAT Solver that implements the most common SAT Solvers strategies, including: CDCL, two-watched literals, restarts and VSIDS. It uses a static divide-and-conquer approach to divide the search space and may be run in several different GPU processors. We intend to take advantage of the massive parallelism brought by this cheap and easy to find environment.

The two works we present, SAT-PaDdlinG and RePaSAT work together, but have their own contributions separately. SAT-PaDdlinG is a SAT-Solver and may be used in many different contexts, as is seen the literature. RePaSAT may use different SAT Solvers and is intended to be implemented in hardware. SAT-PaDdlinG was created to provide

massive parallelism to RePaSAT, but its individual contribution cannot be ignored.

4.1 Structure

SAT-PaDdlinG is a parallel SAT Solver developed for CUDA-enabled NVidia GPUs. It was developed using CUDA C++ and uses object-orientation as its main programming paradigm. Since CUDA C++ allows the programming of both CPU (regular C++) and GPU code, the entire software was built in this language.

SAT-PaDdlinG is basically composed of objects implementing independent SAT Solvers. Each object solves the formula and it receives assumptions along with the formula to be solved. In the same object, the solving procedure may be called more than once, with different assumptions, though the formula cannot be modified. Assumptions are assignments to variables received before the SAT Solver starts solving. They work as decisions that are never undone. These assumptions are used by SAT-PaDdlinG to set guiding paths, used to split the search.

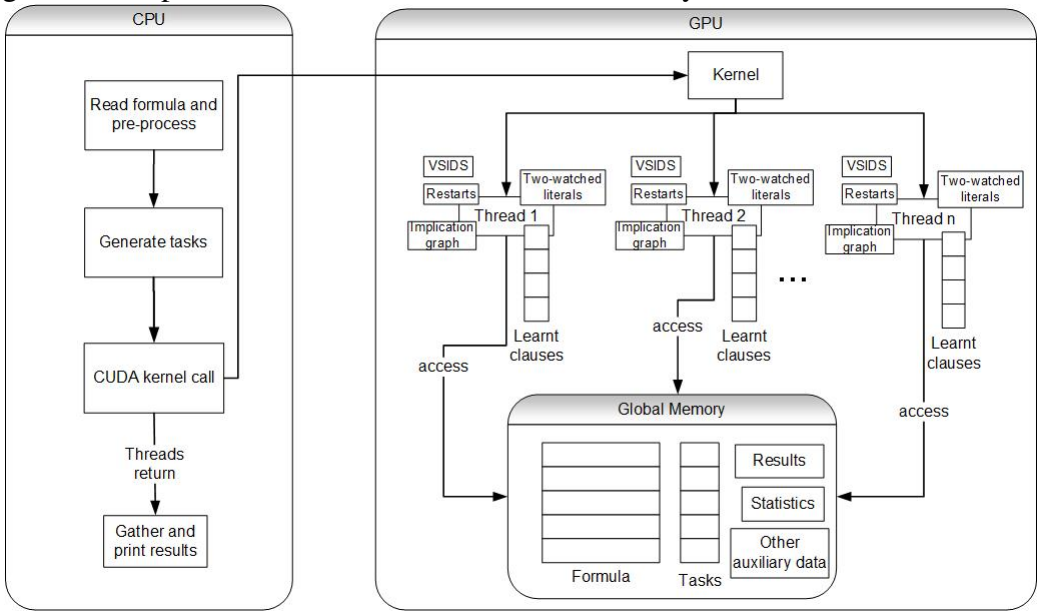
Each object has the capability of resetting its structure, once it is done solving, undoing the decisions, implications and assumptions, setting all variables as free and ready to be reused. After resetting, some data is kept, such as learnt clauses. Such data is only formula-dependent and remains valid even after removing the assignments. Since this object will always be used to solve the same formula, this data remains valid and helps in future searches.

SAT-PaDdlinG uses statically generated guiding paths, without the open and close concept (see Section 2.4.2.1.1), to split the search-space and those independent SAT Solvers objects to solve them, as tasks. A shared list stores these guiding paths and each thread chooses one task from the list at a time and solves it. Once a thread is done solving the task, it chooses another, for there may be more tasks than threads, to try to cope with workload balance issues. Figure 4.1 illustrates the software's structure.

As it is shown in Figure 4.1, each thread has its local data and also accesses shared data structures, such as the formula and the tasks to be solved. The formula is read-only and does not require any mutual exclusion strategy. The tasks are not removed once they are solved, instead, there is a pointer to the next unsolved task that is atomically increased as the task is chosen, making it also read-only (apart from the pointer). It is guaranteed that no two threads will solve the same task.

This structure guarantees minimal communication, hence it is better suited for a massively parallel machine. The threads only have to compete for a task, compete only for one atomic operation that they cannot run together and then in the end for writing the results. All other shared structures either are read-only or ensure that threads write

Figure 4.1: Representation of the execution and memory allocation of SAT-PaDdlinG.



in different vector positions, not requiring mutual exclusion. Communication is generally the main factor for low scalability in parallel applications and it is not an issue for SAT-PaDdlinG.

SAT-PaDdlinG carries out conflict analysis through an implication graph, as well as clause learning and non-chronological backtracking, all as described in (SILVA; SAKALLAH, 1997). It uses two-watched-literals to improve BCP, as described in (TEIGE; HERDE; FR, 2007). It uses VSIDS as the decision heuristic and implement restarts.

4.1.1 Basic Search

As mentioned before, SAT PaDdlinG has objects of independent SAT Solvers, which solve the problem. These independent SAT Solvers start to solve the formula given the assumptions. The algorithm used in this search is much like the Algorithm 1 presented in Section 2.2.2. It, however, before starting solving the problem, preprocesses the assumptions, generating implications from them and checking if they alone solve their part of the problem.

As the preprocessing step, it runs BCP to get implications from these assumptions, that are also never undone throughout the search and are only deleted after resetting the object. Both the assumptions and their implications are given the decision level 0. That is useful, because if the SAT needs to undo an assumption or its implication to make the formula true, it need to backtrack to level 0. However, in such instance, the SAT Solver returns 'unsatisfiable', as shown in Algorithm 1, which is the expected behavior, since the assumptions must not be undone.

SAT-PaDdlinG always keeps a list of decisions, a list of implications, a list of assumptions and a list of free variables. It also keeps the current assignments for the variables in a vector, ordered by the variables' number. Though the assignments could be retrieved by searching in the decisions, implications, assumptions and free variables, this vector is used to speed up this process, allowing it to have a constant complexity for accessing the current assignments.

SAT-PaDdlinG provides many common algorithm and choices of algorithms. It allows as the decision heuristic both VSIDS or a simple choice of the last free variable (low overhead), shown in Section 4.1.2. It allows to used two-watched literals as the BCP procedure, as shown in Section 4.1.3, although it also allows a simple iteration through all clauses, which is more time-consuming, but less memory-consuming. SAT-PaDdlinG carried out conflict analysis through implication graph, as explained in Section 4.1.4 and learns clauses 4.1.4. SAT-PaDdlinG also allows the use of restarts, shown in Section 4.1.6.

4.1.2 Decisions and VSIDS

SAT-PaDdlinG has two different kinds of decision procedures. The first is a basic selection of the last free decision. This was implemented to provide a simple decision for the first versions of this software. The last free decision is chosen, instead of the first, because it works better with an array of free variables.

The second is the VSIDS heuristic. As explained in Section 2.3.5, it is a dynamic algorithm that enables a smarter choice of decisions. To implement this algorithm, initially we used a simple array containing the sum of literals. Many SAT Solvers, such as MiniSAT, use a heap to improve the time to retrieve the literal with highest sum, from $O(n)$ to $O(\log n)$ (KLEINBERG; TARDOS, 2006). We have started implemented this algorithm with a heap, but it is not finished yet.

A heap is capable of returning the decision with highest sum in logarithmic worst time (KLEINBERG; TARDOS, 2006). However, since we need to constantly update the literals' sum, after every conflict, we need to fetch literals out of their sum order, given their variable's number, which in a heap can only be done in linear time, since the variable has nothing to do with the key used to organize the heap: the literals sum. Since we use a heap to avoid linear time in the first place, this must be dealt with.

For that reason, we also intend to implement (and started implementing) an index vector. Every index position of this vector refers to the position in the head of the variable of number equal to this index, each position of this vectors holds two pointers, one to the positive literal of the current variable, and the other to the negative one. With this vector, we are able to fetch literals out of order in constant time, and update in logarithmic time, since both removal and insertion are logarithmic in the heap (KLEINBERG; TARDOS,

2006).

4.1.3 BCP and Two-Watched Literals

The implementation of this strategy was done mainly using (TEIGE; HERDE; FR, 2007) as reference, although the mapping to variables was not implemented as defined, since the only variables we use are atoms in the Boolean formula. This strategy includes both a data structure and algorithms to deal with it. The objective of this strategy is to speed up the BCP process.

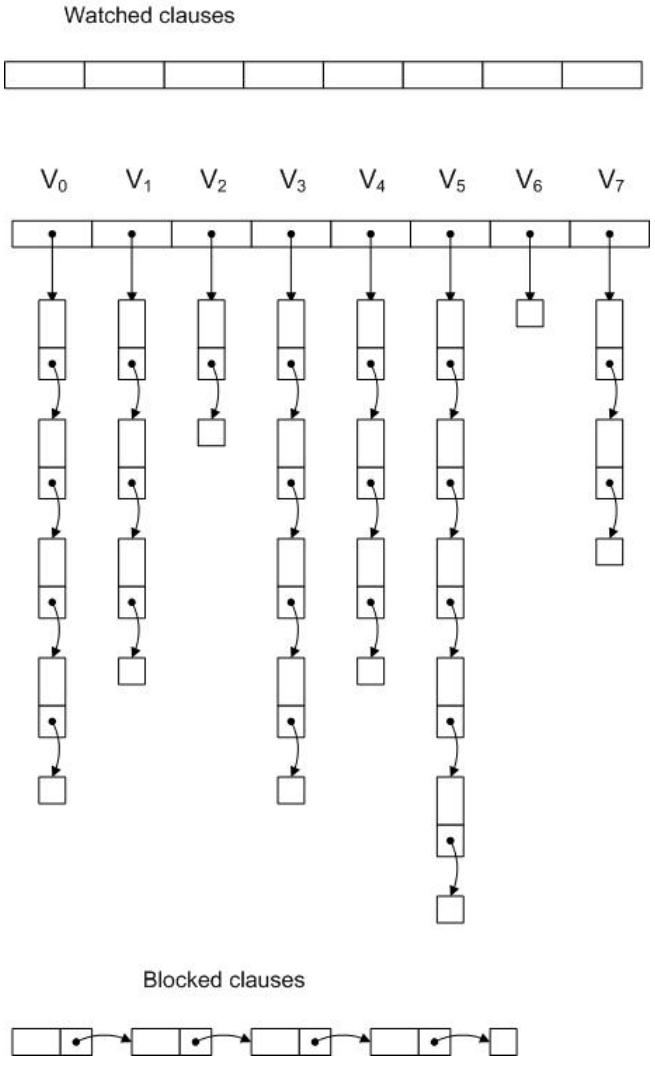
Up until the implementation of this strategy, after each decision or implication, the SAT Solver scanned the entire clause database looking for unit clauses (to find implications) or conflicts (to know when to backtrack). The data structure used by the two-watched literals strategy holds a list of clauses for each variable. Each clause has two different watched literals, either free or satisfied (consistent), and the list of the variables of each of these two literals holds a reference to the clause. When a decision is made, only the clauses in the corresponding variable list are visited, because these are the only clauses that can actually cause an implication or a conflict. Notice that there must be two different watched literals per clause, so unary clauses cannot be processed with this strategy and must be preprocessed.

To understand why only clauses containing watch literals of the analyzed assignment must be visited, consider two scenarios: a clause does not contain the literal of the decision (or implication) and the clause contains it, but it is not watched. It is easy to see that a clause that does not contain the variable of the assignment cannot cause implications or conflict, since it will not have its state changed. A clause that does contain the variable but has two other watched variables also cannot clause implication or conflict. The clause must be unit and unsatisfied, respectively, to cause those states, requiring exactly one and zero non-unsatisfied literals, respectively. Since the two watched literals are not modified and are either free of satisfied (because they are watched), the clause's state is not modified and it does not need to be visited.

This strategy greatly improves the efficiency of a SAT Solver, by allowing it to visit only a portion of the clauses per iteration, as opposed to all of them. SAT-PaDdlinG implements this scheme and its data structure is depicted in Figure 4.2. It is basically a vector of linked lists. The vector's size is exactly the number of variables and each list contains the clauses that watch a literal of that variable. When a literal starts being watched in a clause, the clause must be added to its list, as well as, when it stops being watched in a clause, this clause must be removed from the list.

The clauses are represented in a struct along with a the index of the two literals in that clause. There is a list of watched literals (not the variables list), just to hold

Figure 4.2: Data structure used to represent the two-watched literals scheme in SATPaDdlinG.



the watched clauses pointed by the variables list. There is a list for every variable that contains a pointer to all the watched clauses that watch that specific variable. There is also a blocked clauses list, which contains the unit (not unary) and unsatisfied clauses, until they have again two consistent literals.

This is a dynamic structure and it changes after every decision, every implication, every assumption and backtracking. If a decision, implication or assumption is not passed through this structure, some of these clauses may end up watching inconsistent literals, which could cause it to miss implications and conflicts. It also needs to be modified after backtracking. When decisions (and implications) are undone, unary and conflicting clauses may start having 2 consistent literals again, which requires them to be watched.

4.1.4 Conflict Analysis

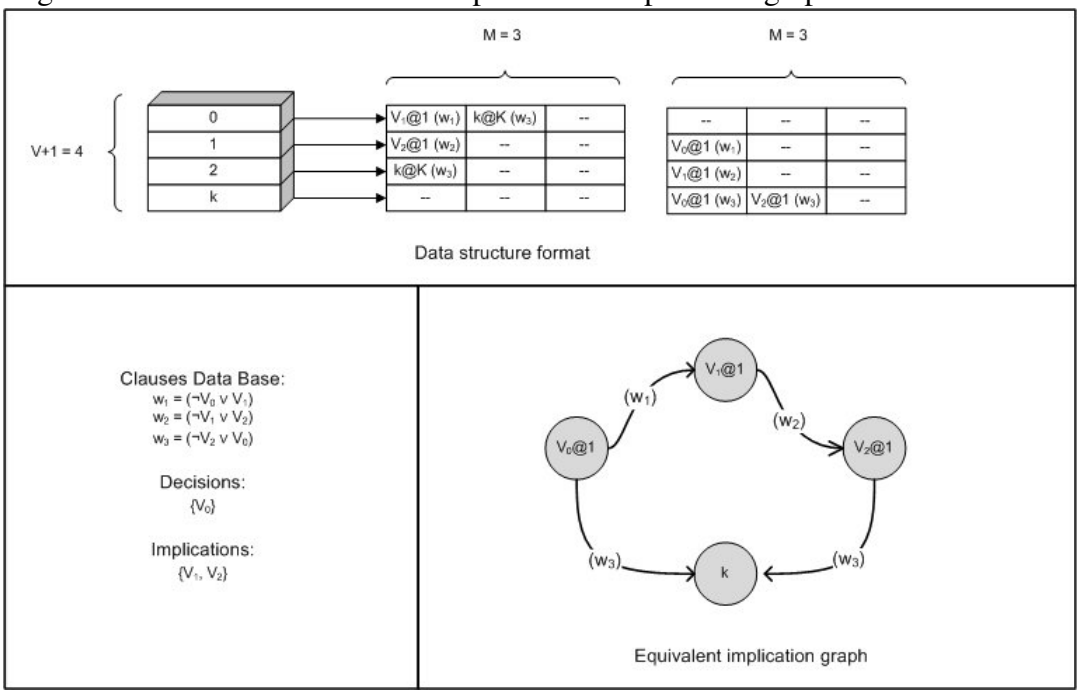
SAT-PaDdlinG has an implication graph for each thread, as shown in Figure 4.1, that stores the decisions, implication literals and assumptions, as well as keeps record of the causes of implications and corresponding clauses. This structure is built as described in literature (SILVA; SAKALLAH, 1997; KROENING; STRICHMAN, 2008) and is used to learn clauses based on current decisions. As described in literature, the clauses rely on decisions to be learnt, but are directly implicated by the input formula and completely independent of decisions (HOLLDOBLER; MANTHEY, 2011). Hence, they do not need to be deleted when the decisions that generated them are undone. They also are independent from assumptions, which are treated as decisions, so they can be used after resetting the object and, in the future, shared among different threads. The data format is shown with an example in Figure 4.3.

In this graph, each vertex corresponds to an assignment of a variable, a literal, and there cannot be more than one literal for each variable in the graph, because we cannot have more than one assignment for each variable. Since the variables are represented by numbers that vary from 0 to $V-1$, we use the variable number as an index for the graph. The vertex of position i corresponds to the variable of number i, whether it is present in the graph or not and there is a flag saying if the vertex is present in the graph. The conflict vertex has index V. Each vertex has a position in a vertex list (an array), as shown in Figure 4.3.

The structure has basically 3 components:

- **Vertices list** : It is the first vector to the left in the upper box in Figure 4.3. The list of vertices contains possible vertices of these graph, included or not. Each position of this list contains information about this vector, including how many edges it contains.

Figure 4.3: Data structure used to represent an implication graph in SAT-PaDdlinG.



- **Forward edges** : It is the second element in the middle. This is a matrix that represent a list of lists. Each line is the list of edges of a vertex and the columns form the list, holding information of the edge. Each line l represent the vertex of the variable l. The columns of this line contain the links between the vertices, holding the number of the variable connected to the vertex as well as the decision level and antecedent clause. They are implemented as a list of edges, for better memory usage. In this implementation, only one conflict vertex (k) is possible.

- **Backward edges** : It is the last matrix to the right. This is analogous to the forward edge, but every line refers to the destination instead of the source and the links are the sources. The backward edges are used in algorithms and are consistent with the forward edges. We implement backward edges, because the clause learning algorithm require a backward search from the conflict. As a matter of fact, the backward edges are the actually used ones, and forward edges are reserved for testing.

In Figure 4.3, the $'V'$ value refers to the number of variables, while the M value is the edge capacity, which must be set before starting the solving. Rather than creating a matrix $V \times V$, each (i, j) position referring to the edge from i to j, we decide to create a list of edges, to consume less memory, with the cost of extra computation. In other words, each line i of the forward matrix is a list of edges from the vertex of variable i (analogous for backward, only the destination is i).

The value $'M'$ is set as the number the occurrences of the most common variable

in the formula (the number of clauses that contain it) or the size of the largest clause, the largest of these values (maximum). In the case the forward edges are not stored (see below), the size of the largest clause is used.

Optionally, the software allows not to store the forward vertices. They are currently interesting for testing, but require an extra space and more computation to keep both matrices up-to-date. The current algorithm that generates a learnt clause for a conflict implication graph only requires the backward vertices, leaving the forward ones with no real purpose. The forward edges are available, in case a different algorithm is used.

4.1.5 Clause Learning

To generate a learnt clause, after a conflict is identified, we have used the algorithm used by GRASP, presented at (SILVA; SAKALLAH, 1997). After it is generated, the clause is learnt and:

1. If the learnt clause (and hence the conflict) is derived from assumptions or the formula (highest decision level of literal is 0), the SAT Solver returns UNSAT.

2. If the clause is unary, the SAT Solvers backtracks to the decision level 1 (KROENING; STRICHMAN, 2008) and adds the only literal in the formula as an implication in level 0. This is what we considered as an implication from the formula, it is flagged as such and it is never undone, not even after resetting the object.

3. It is not unary and the two highest literals have different decision levels, the SAT Solvers backtracks to the decision level of the second highest (without undoing it) and implicates the highest literal, as described in (KROENING; STRICHMAN, 2008) and done by Chaff (MOSKEWICZ; MADIGAN; ZHAO, 2001).

4. It is not unary and the two literals with highest decision level have the same decision level. Currently, since the learnt clause is an asserting clause, this state returns an error, for it must never occur. If the clause learning algorithm is ever changed to one that may generate clauses that are not asserting, it must backtrack to the level of its largest literal, undo the two or more literals in this level, and start making decisions from it.

To explain, the first step is done because conflict implicated from the formula or assumptions imply that the problem (with the specific assumptions) is UNSAT. As for the second step, it was implemented as described in (KROENING; STRICHMAN, 2008; MARQUES-SILVA; LYNCE; MALIK, 2009) and implemented by Chaff (MOSKEWICZ; MADIGAN; ZHAO, 2001) as an improvement to the basic non-chronological backtracking proposed by GRASP (SILVA; SAKALLAH, 1997). The distinction between the steps

3 and 4 was implemented because of the expected behavior when learning an asserting clause. If the learnt clause is asserting, as it is implemented today, the option 4 is not possible to happen, but it is necessary to be analyzed in case the algorithm for clause learning is changed.

4.1.5.1 Learnt Clauses Management

Because GPU memory has generally less capacity than the main memory in a computer, SAT-PaDdlinG carries out a management of its clauses, deleting with some frequency to avoid a number of clauses that may consume all memory. A value is set for clauses capacity, which we originally have set as 20, but can be changed. The choice for deleting a clause, when this threshold is reached is: the oldest clause is deleted. To accomplish that, we keep a pointer to the first clause. When the list of clauses is full, we delete the clause the pointer is pointing at and increment the pointer. When the pointer reaches the end of the list, it goes back to the first position. Since clauses are added in this order as well, we end up removing the last added clauses.

This policy for management of clauses was chosen to try to leave the most relevant clauses, while still not adding too much overhead. If the SAT Solver is solving a specific portion of the search space, if it deletes recently learnt clauses, it will most probably delete relevant clauses to that portion. On the other hand, older clauses are more likely to be less relevant. A possibly better approach would be to keep a counter of clauses that cause conflicts, which would imply clause that are more active. This is possible improvement to be added in the future.

Whenever a clause is removed, it is necessary to also remove it from the watched-literals list. In this part of the code, we do not have the information of what literals in the clause are watched, so the list of every literal in the clause is searched and once one is found, we can automatically determine the second watched literal, which is then directly removed from its list.

4.1.6 Restarts

SAT-PaDdlinG implements restarts. Due to its simplicity, we implemented a configurable geometric restart, in which a configurable initial number of conflicts is set, which defines the number of conflicts necessary before restarting, as presented in (HUANG, 2007). This value is updated after each restart by a factor, which is also configurable. For instance, if the number of conflicts is 100 and the factor of increase is 1.1, after the first 100 conflicts, SAT-PaDdlinG restarts and the number of conflicts is updated to 110, which is the old 100 multiplied by 1.1.

4.1.7 Mutual Exclusion

Because the objects of SAT Solvers used by SAT-PaDdlinG are completely independent, not even implementing clause sharing, they do not need any kind of mutual exclusion. Mutual exclusion is, however, necessary when getting a task from the list of tasks, which is shared by the threads, as well as when writing the results, for the unlikely possibility that two objects find an answer at the same time.

Both these cases are resolved using atomic operations. Rather than using critical sections, that may make idle threads, we simply used operations that are guaranteed by CUDA to be executed atomically and since these are single operations, threads do not wait for long. In the case of the task, we kept them in a vector and a pointer, in fact an integer value indicating the position of the next unsolved tasks, points to the next thread. It initialized as zero. When a thread attempts to get a tasks, it atomically increases the value of this counter, such operation returns its previous value. It gets the previous value to select the task and leaves the counter's value set for the next thread.

The results use a similar approach. A 'guardian' variable is used. Its value is set and a thread is only allowed to write the results if this variable contains its initial value. When a thread attempts to write a result, it atomically modifies this variable's value, by incrementing it, as well as gets its previous value. If its previous value was the initial one, it is allowed to write the result. This guarantees that only one thread will write the result and since we only want a single solution for the problem, this gives the expected behavior.

4.2 Experiments

We have carried out some experiments with SAT-PaDdlinG. The objective of these experiments is to show different characteristics about SAT-PaDdlinG and generalizations about execution of a DPLL SAT Solver on a GPU environment. Each subsection in this section presents a different experiment, with a different conclusion to be reached. The experiments are:

- **Speedup experiment** : This experiment has the objective of showing the parallelism gain brought by SAT-PaDdlinG. Since SAT-PaDdlinG was developed in a GPU, aiming at taking advantage of parallelism, we first analyze if, in fact, this advantage exists. It is important to point out that this experiment used tasks generation in both sequential and parallel execution, not to have more than one variable tested. This experiment is shown in Section 4.2.1.

- **Slowest against average thread behavior** : This experiment was carried out aim-

ing at showing a characteristic we have identified in SAT PaDdlinG. SAT-PaDdlinG seems unable to accelerate a few inputs and we have identified that the problem lies on workload distribution problems. This experiment shows strong evidences to this conclusion and is presented in Section 4.2.1.

- **Threads X Blocks**. This experiments allows us not only to make very interesting conclusions about SAT-PaDdlinG, but also about general DPLL SAT Solvers on a SIMT. Since hitherto no DPLL SAT Solver was fully tested on a GPU environment, with a SIMD characteristic, its behavior is unknown. This experiment shows an interesting behavior and a serious limitation when it comes to DPLL SAT Solvers on SIMT environments and the difference between using blocks and threads for parallelization. It is presented in Section 4.2.3.

- **Blocks Limit**. Yet another limitation of SAT-PaDdlinG is presented in this experiment. Possibly because of scheduling in the GPU, we have identified some limitation when it came to the number of blocks. This experiment is presented in Section 4.2.4.

- **Parallelization Overhead**. Because we divided the problem in many tasks to solve the problem, we have identified some overhead concerning this split. We present in this experiment some empirical data concerning this overhead. This experiment is presented in Section 4.2.5.

Since all experiments were set up to be run in SAT-PaDdlinG, our basic inputs for them are CNF formulas. We have CNF formulas from 3 different sources: benchmarks for experiments; ToughSAT, an automatic generator of tough SAT formulas and formula generated with Ursa. We believe that, because we have inputs from these three different sources, we have been able to comprise different types to formulas to identified different kinds of behaviors. These are the sources and the inputs generated from them:

- **Benchmarks**: In this case, the CNFs were acquired from benchmarks. They are: hole6 (42 variables, 133 clauses) and aim-100-1_6-no-1 (100 variables, 160 clauses).

- **ToughSAT** (YUEN; BEBEL, 2014): ToughSAT is a project to generate hard SAT instances. The instances are: subset_random_4_5 (63 variables, 194 clauses) and factoring_6_13 (56 variables, 226 clauses).

- **Ursa** (JANICIC, 2010): As presented in Chapter 5, Ursa is a tool that automatically generates (and solves) CNF formulas from imperative/declarative code. The generated instances were: QUEENS (9,8) (225 variables, 1110 clauses) and TS (3,8) (493 variables, 1568 clauses), respectively, the problem to place 9 non-attacking queens in a 9x9 chess board and the Traveling Salesman Problem with 3 cities.

The experiments used statistical data generated by SAT-PaDdlinG itself and the 'time' tool available in Linux distributions. The metrics used to analyze this tool were:

- **Total application time** : Data from the 'time' tool, that returns the elapsed time between the beginning of the application, when it was run, to the end. Because this was retrieved by a tool running outside the software, it may include time to launch it, which we consider insignificant.

- **Speedup** : Speedup is calculated as the ratio of the application time (or other time, if it is the case) for a sequential execution and a parallel execution (JURKOWIAK; LI; UTARD, 2005). This metric is widely used to check the improvement brought by parallelism (JURKOWIAK; LI; UTARD, 2005) and shows how many times the application has become faster by parallelizing it. Given p processing, units solving the problem, ideally, the speedup would be p (JURKOWIAK; LI; UTARD, 2005), this is called the ideal speedup. However, such a scenario is rare (JURKOWIAK; LI; UTARD, 2005), and the speedup is generally smaller than p. If the speedup is greater than p, it is called superlinear speedup (JURKOWIAK; LI; UTARD, 2005).

- **Average thread/block time** : This metric was calculated collecting the time each threads or blocks took to solve the problem and calculating the average. This time only includes the actual time each thread used to run and CPU time, time to run the constructors of classes and writing the results when found were left out. This is a good metric to see the behavior of the threads as a whole, but is does not necessarily describe the behavior of the software as a whole.

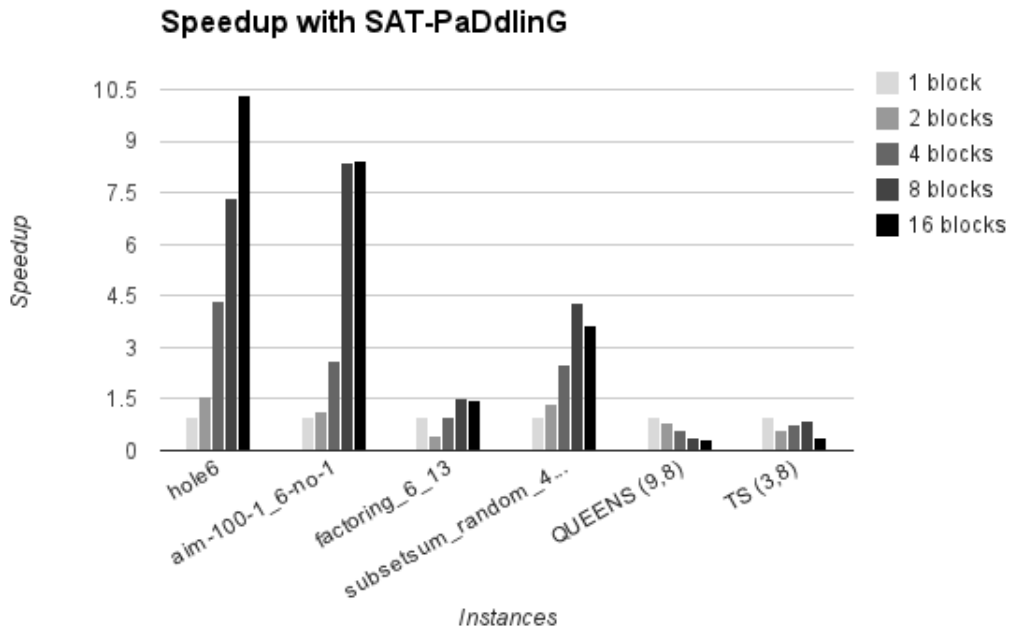
- **Slowest thread/block time** This metric is somehow similar to the previous, only instead of calculating the average of the threads, it returns the time the slowest thread took to finish executing. As the previous metric, it only includes actual time to solve the problem and empirical data show that the application as a whole behaves as this metric, that is, this metric represents well the behavior of the software. It makes sense, since the slowest thread holds the problem down and CPU time and other execution time not include in this metric are too small when compared to the execution of the threads.

4.2.1 Speedup Experiments

In this experiment, we analyzed the speedup provided by SAT-PaDdlinG. Breaking the problem into tasks, in many cases, have caused a parallel overhead. This behavior is analyzed in Section 4.2.5. In this section, we intend to present the scalability of SAT-PaDdlinG, regardless of its overhead for parallelization.

We decided not to compare with a sequential execution without tasks, to avoid

Figure 4.4: Instances' speedup.



two different variables changed in the same experiment: number of blocks and number of tasks. Thus, the number of tasks is fixed at 128. All experiments shown here were executed with an increasing number of blocks, while the number of threads per block was fixed in 1, experiment in Section 4.2.3 explains why.

The speedup metric is used, calculated with the total application time. This allows us to see the improvement in the execution as a whole, not only some parts of the software. The speedup is presented in Figure 4.4.

The "TS" and "QUEENS" problems have not reached speedup greater than 1, showing no improvement in the parallel version. The 'factoring_6_13' input showed some rather low scalability, while still improving up to 8 threads. The input "subset-sum_random_4_5" showed some interesting improvement, though still far from the ideal speedup. The input 'hole6' has showed the largest speedup, reaching a factor of 7 for 8 threads and more than 10x for 16 threads, greatly improving its time. The instance 'aim-100-1_6-no-1' has also shown high speedup. Benchmark instance problems seem to be fitting to SAT-PaDdlinG.

We have reached high-speedup for some cases and some lower but still interesting speedup for others. For cases with low speedup, we have identified bad workload distribution, as we will show in Section 4.2.2. Even though we have broken the problem in many parts, literature has long established that static division of the search space causes bad workload distribution (MARTINS; MANQUINHO; LYNCE, 2010).

Though we were unable to improve every input, we can see that this approach

does provide significant improvements in some cases. We had a speedup greater than 10 and another greater than 8, meaning that we have been able to cut the time by these factors. If, by any chance, the 'hole6' instance took a month to solve, by running with 16 blocks, it would take less than 3 days, a very significant improvement. Of course, since we have much room for improvement in SAT-PaDdlinG, we expect to have this behavior happening for more instances.

It is also important to point out that even instances with a speedup that is not as high, if they reach a speedup higher than 1, it compensates parallelizing. GPUs are cheap and easy to find environments, generally idle when a computer is not running graphical applications. For example, in "factoring_6_13", simply by using a GPU, we can run in with a speedup of 1.5. That means that if a sequential approach to a month to finish, the parallel version would take 20 days. A wait of 10 fewer days.

4.2.2 Slowest Against Average Thread Behavior

As mentioned before, workload problems have been reported in literature for static search-space split. Low speedup is generally associated with communication. However, in our case, the resolution processes are independent and communication basically non-existent. To identify these workload problems, we compared the time of the slowest block time against the average block time. These metrics are presented in the beginning of Section 4.2. The slowest thread indicates the behavior of the software as a whole, since it can only finish after this slowest thread is done.

The essence of this experiment is to compare these values for the results with decreasing speedups. If both show slow improvement as the number of threads increase, that shows that the threads were affected as a whole, indicating idle threads. On the other hand, if the average has an increasing speedup, differently from the slowest, that shows that the difference in their time is causing the slowdown, so the threads as a whole must be improving. This indicates workload problems, since we have very fast threads, along with the very slow ones that stabilize the average.

To better understand this, imagine any possible factor, such as communication, that is slowing down a thread. This factor affects this thread, if this is the worst thread, the worst thread time is affected. In the same time, if this slowdown is not compensated by other threads improving, which would slow down more than one thread, the average time is also affected making the average time to solver per thread to be greater.

On the other hand, workload distribution is different. When we distribute tasks among threads, the ideal scenario is that each task would take S/t time to solve, where S is the sequential time and t the number of threads, ignoring parallelization overhead of this logic. If we badly distribute a problem among threads, there are threads that take

longer to solve than the ideal and there most certainly will be threads that take less time than average, balancing the average. For that reason, if we have workload distribution problem we expect that average thread time to have a different behavior from the the slowest thread time, improving over time.

Figure 4.5 shows those metrics for the two problems with decreasing speedup, QUEENS and TS. These instances were generated by Ursa. In Sections 5.4.1 and 5.4.2 we have shown that these instances generally do not parallelize well, which seem to be a characteristic from instances generated by Ursa. That explains why these were the worst presented here and this experiment will also allow us to understand this behavior, at least in regard to SAT-PaDdlinG.

One can see that, for the Queens Problem, the worst thread time increases as the number of threads is increased. For the Traveling Salesman Problem, it fluctuates a little, but in general also increases. On the other hand, for the average time, it actually improves in both cases up to 8 threads, and continues improving up to 16 thread in the case of the Traveling Salesman Problem. This large discrepancy can be explained due to the bad workload distribution. This indicates that our approach can be further improved with a smarter strategy from splitting the problem, such as dynamic search-space split, which improves workload balancing.

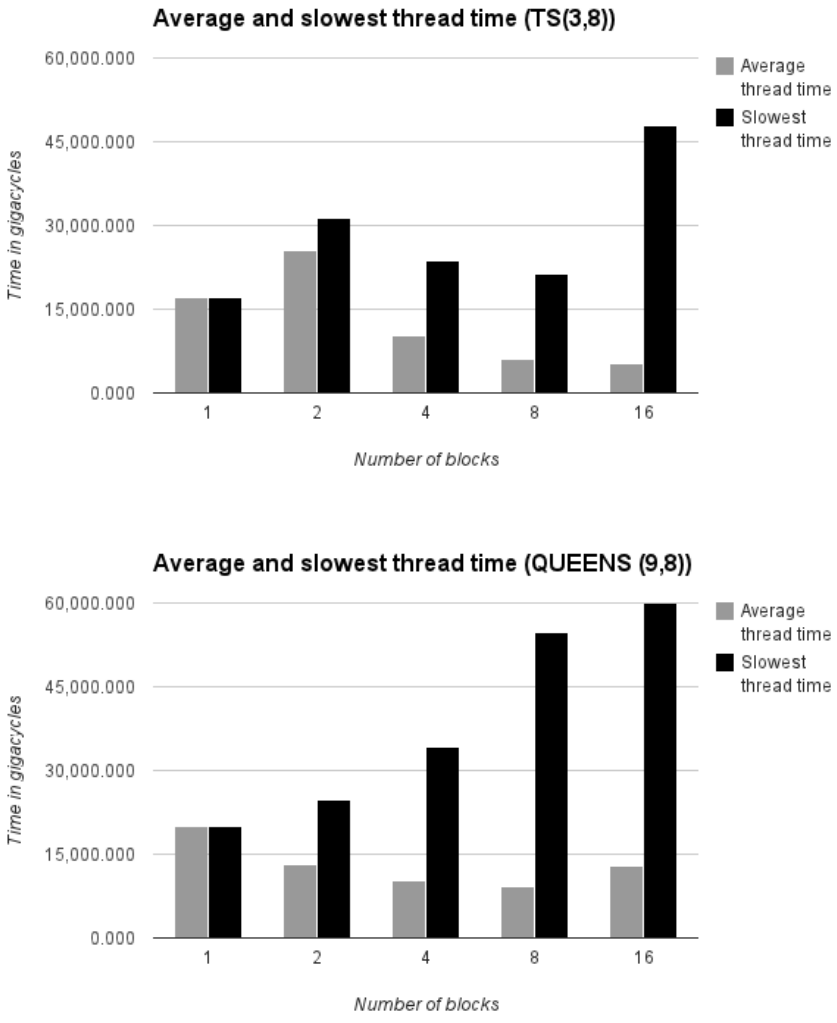
This conclusion together with a well established understanding in literature that statically dividing the problem brings bad workload distribution indicates that we could improve this instances by providing a better workload distribution of the problem. Thus, our next step is to implement a dynamic search-space split strategy to reduce the discrepancy between the worse thread time and the average. A proposal for a dynamic search-space split is presented in Section 4.3.1.

Also, when attempting to parallelize instances generated with Ursa in RePaSAT (Chapter 5), we have been unable in most cases. This experiment indicates that the problem with these instances in SAT-PaDdlinG lie on workload distribution, they may have some structure that is not easy to distribute. However, the portfolio approach used by ManySAT was also no capable of parallelizing them, as shown in Section 5.4.2, which does not break the problem in parts. The instances structure in regard to parallelism must be further studied to allow better conclusions.

4.2.3 Threads X Blocks

In Section 2.5.1.1 we presented a few characteristics of CUDA regarding the scheduling of threads and its SIMT structure. When creating a block with many threads, groups of 32 threads are created, threads from the same block. These groups of threads, called warps, are run with a sole program counter, forcing every thread in that warp to

Figure 4.5: Difference in average thread time and longest (max) thread time for queens and traveling salesman problem.



run the same instruction and become idle if the execution is in a different path from the path they should follow, because of a branch. Threads from different blocks are created in different warps. These concepts are shown in Section 2.5.1.1.

That is very useful for many applications in GPU. If we want to run the same operation in different parts of a vector, in a SIMD model, we could simply create these light-weight threads and run the operations. The question that arises is: How does this behave with a SAT Solver? Can we take advantages of these warps, using all 32 threads per warp or are we forced to have only one thread per warp, making the other 31 threads inactive? All 32 threads are always created, if the execution has fewer threads per warp than 32, rather than not being created, the others are created and become inactive (CUDA C Programming Guide, 2013). This is a waste of possible parallelism that is already created and ready to be used. The CUDA C Best Practices Guide (Cuda C Best Practices Guide, 2013) advises to use a number of threads per block multiple of 32.

The behavior of a DPLL SAT Solver changes a lot, depending on the part of the formula it is solving. It may or may not find a conflict and learn a clause. It may or may not have to backtrack and may backtrack to different levels. It must analyze different clauses in BCP, when using two-watched literals, depending on the clauses in which the decision is watched, and their numbers, sizes and states may be completely different. It may or may not generate implications and those will most certainly be different in most cases for different decisions. The threads are bound to have very different behaviors.

In spite of that, most probably there will be instructions in common. The whole solution process is a loop, which always includes decision making, BCP, checking if a solution is found, etc. Many threads in the same warp should not be as efficient as the same many threads in different warps, capable of running those instructions independently and never becoming idle while waiting others to finish their instructions, but that is not to say that they would not be better than sequential execution. In a sequential execution, one task would have to "wait" until its previous were solved to start being solved, while in parallel with threads within the same warp, there would be some waiting, but also some shared instructions. That is, at least, an expected behavior.

Because of that, to understand the real implications of using threads within or outside a warp, we show some experiments that compare this behavior. An empirical analysis is an interesting way to check these two approaches: threads within the same warps or in different ones; in a way to include the possibly missing variables of a CUDA-enabled GPU and the software as a whole. We are actually the first people to check the viability of running a DPLL SAT Solver in a SIMT environment.

In this experiment, we have calculated the average time per thread for solving the problem. The time is shown for the instances when increasing the number of threads per

block, while keeping the number of blocks set as one and then increasing the number of blocks with one thread per block. Having at most 32 threads within the same block is the way to force them to be in the same warp, while placing them in different blocks force them to be in different warps. The sequential execution is also shown. The number of tasks used is this experiment were 128 and sequential execution also used this number of task not to change this variable.

In the end, we identified when working with SAT-PaDdlinG that we could not increase the number of threads per block to provide parallelism. Because CUDA operates in this SIMT fashion, running the same instructions per thread seem to completely compromise gains in parallelism. As mentioned, since each thread has a different SAT Solver in a different path and SAT Solvers tend to have different behaviors, when using more than one thread per block, we found out that that actually decreases the time. Figure 4.6 illustrates.

We can clearly see that by increasing the number of threads, the time has increased as well. That shows us that by using threads we do not achieve improvements in parallel. The input 'aim-100-1_6-no-1' is the only one that has a decreasing time when using increasing threads per block, but still by using threads we have a much worse execution than the sequential one, with no gain. The 'hole6' input was the only one in which an execution with threads had a better time than sequential, but it was a very small improvement and was only for 2 threads, become worse again for 4 and more.

Since all inputs but 'hole6' became worse when using threads and for all but 'aim-100-1_6-no-1' the time increased as the number of threads increased, we conclude that by using the used of more than one thread per block we are unable to take advantage from parallelism. If not even in average these problems can be solved with threads, let alone considering the worst time. Also, in every case, increasing the number of threads per block was worse than increasing the number of blocks.

Figure 4.7 shows the comparison made in Figure 4.6, only arranging by instances, not number of threads and blocks and showing only for 16 threads or blocks. We rearranged them this way to make is easier to compare different instances. Because the size of the instances vary, we have grouped in two different graphs, to have instances in similar scale in the same graph and make it more visible.

Our ultimate conclusion is that DPLL SAT Solvers cannot take advantage of an SMIT behavior, when parallelizing by splitting the input. The division of the problems within the same warp does not allow the software to improve from the parallelism. In fact, it is advised with high priority in (Cuda C Best Practices Guide, 2013) to avoid different execution paths within the same warp, something that is not feasible to do with conflict analyzing and backtracking.

Figure 4.6: Comparison between execution time by increasing the number of blocks and increasing the number of threads per block. Sequential execution is also shown.

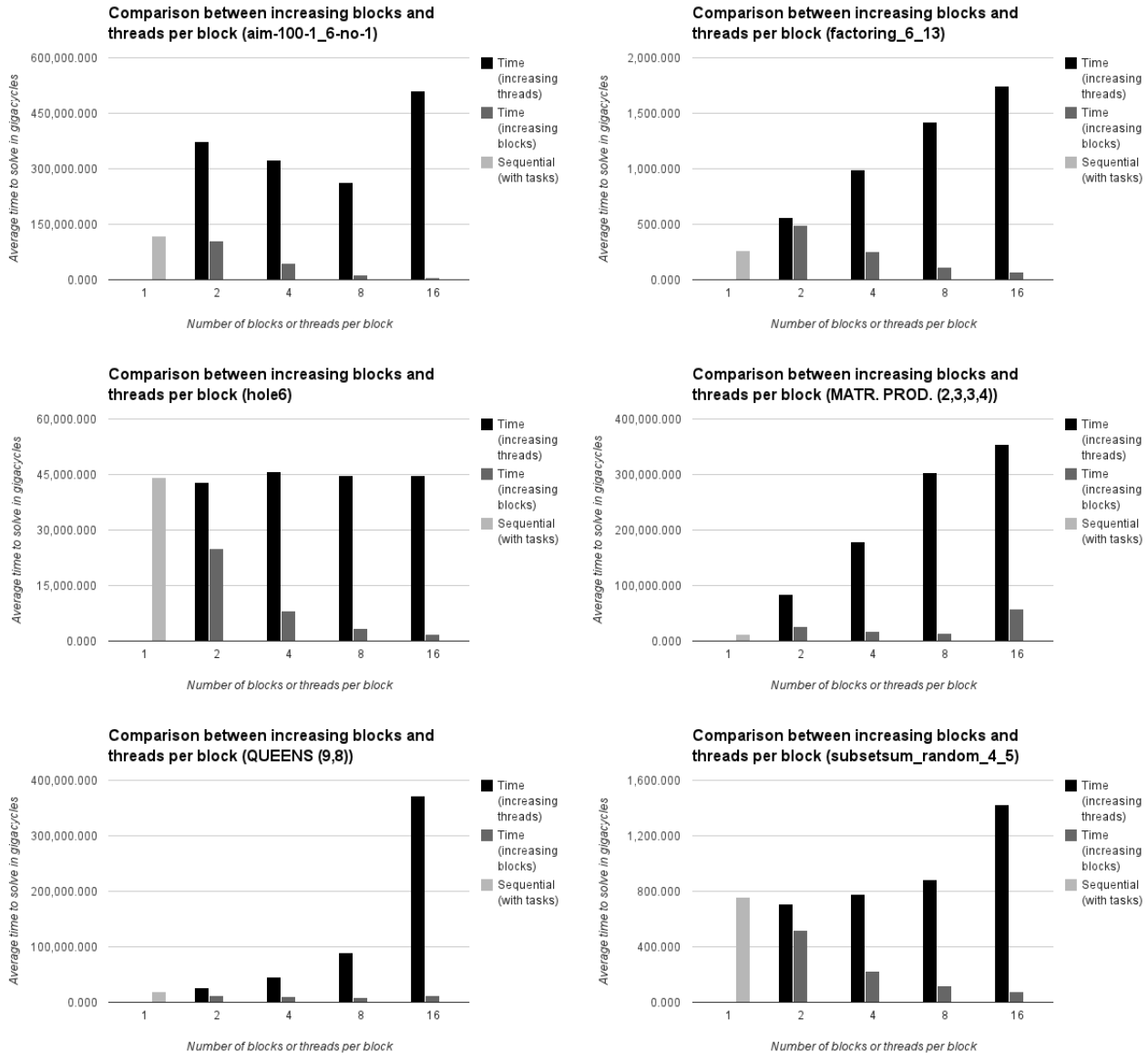
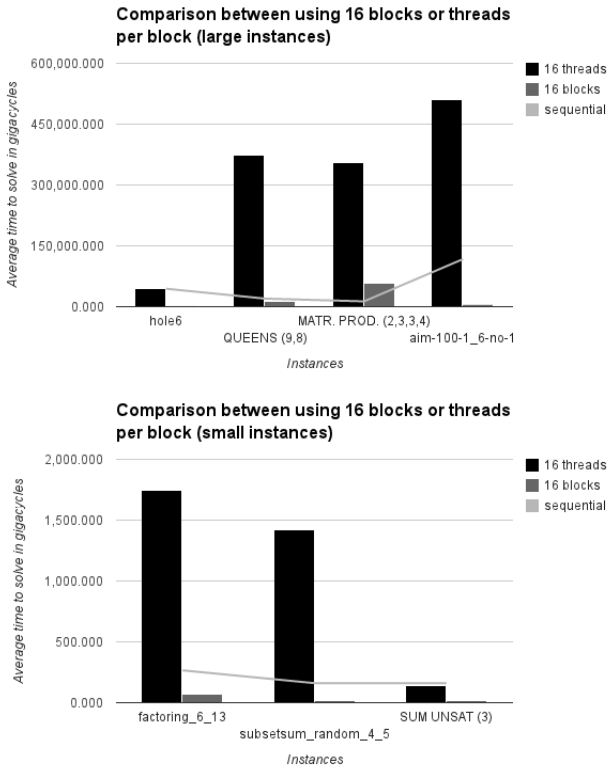


Figure 4.7: Comparison with 16 threads and 16 blocks, grouped by instance. Line represent sequential run.



Though this behavior was known before, DPLL SAT Solvers were never analyzed in this environment and we found out that we are able to take advantage from it, as long as we keep the running threads in different warps. In Section 4.3.2, we propose a different type of parallelism that we intend allow us to take advantage of the parallelism brought by threads in the same warp, still allowing the current parallelism to exist.

4.2.4 Blocks Limit

After some tests, we have found a limitation in our experiments with CUDA. Our experiments showed that there is a limit of how many blocks can be run. When running with 16 blocks, all blocks were able to run and solve some part of the problem. However whenever we increased this number to 32, only 16 of the blocks actually got some jobs to run, the other did not run any jobs. The Figure 4.8 shows a graphic of the individual blocks time for the problem hole6.

We can see in Figure 4.8 that the first 16 threads (from 0 to 15) had a significant execution time, while the others did not have any significant time. Though we showed only for one example, this has repeated in every example we have tested. After analyzing, we found out that all blocks were executed, because they run the constructors of the basic classes. We, then, analyzed the number of solved jobs, depicted in Figure 4.9.

Figure 4.8: The individual blocks total time for an execution with 32 blocks for problem hole6.

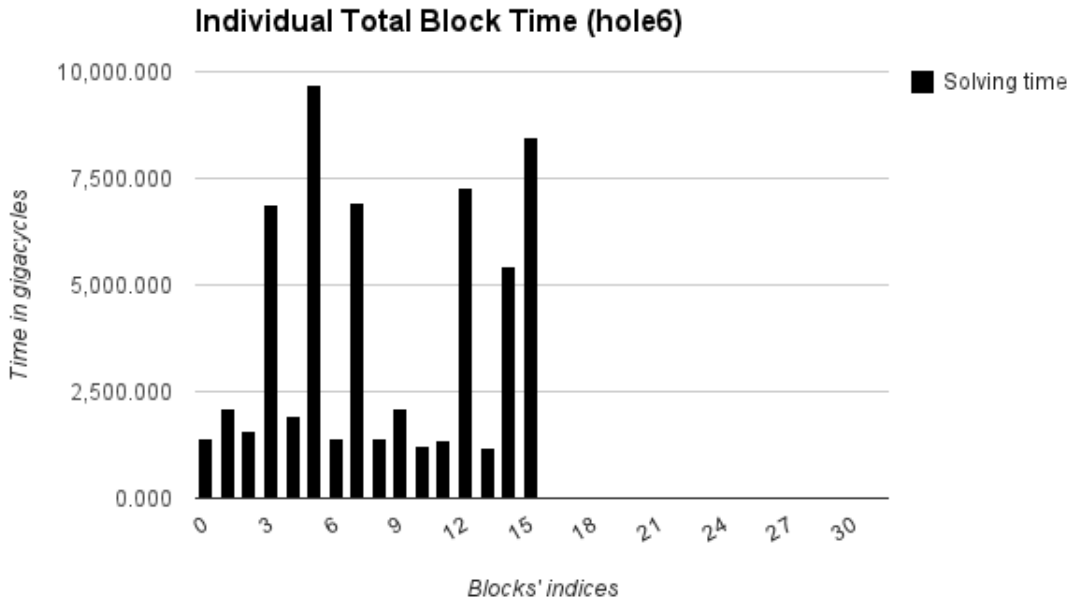


Figure 4.9: The number of solved jobs per block.

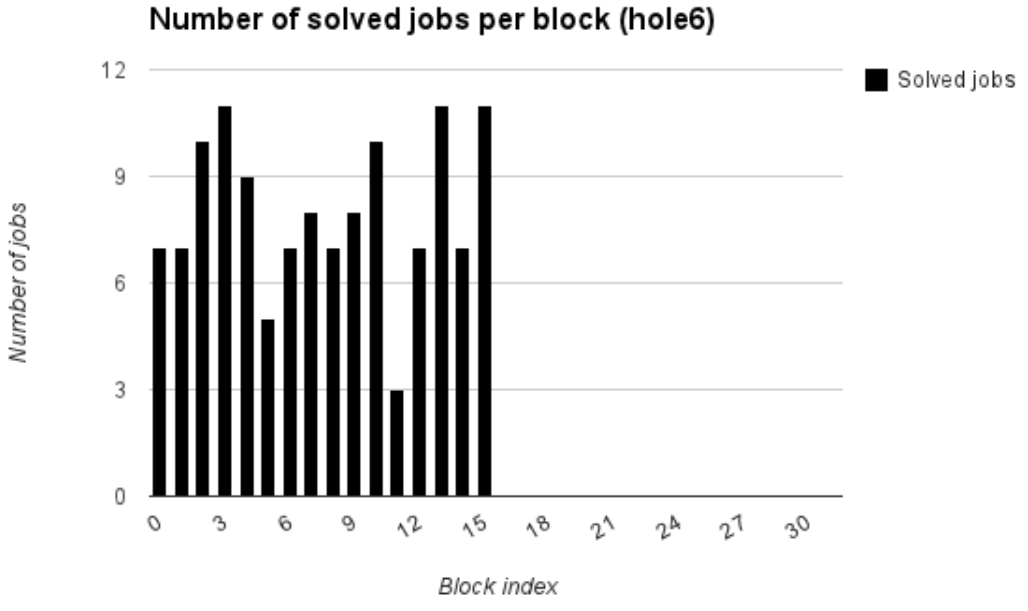
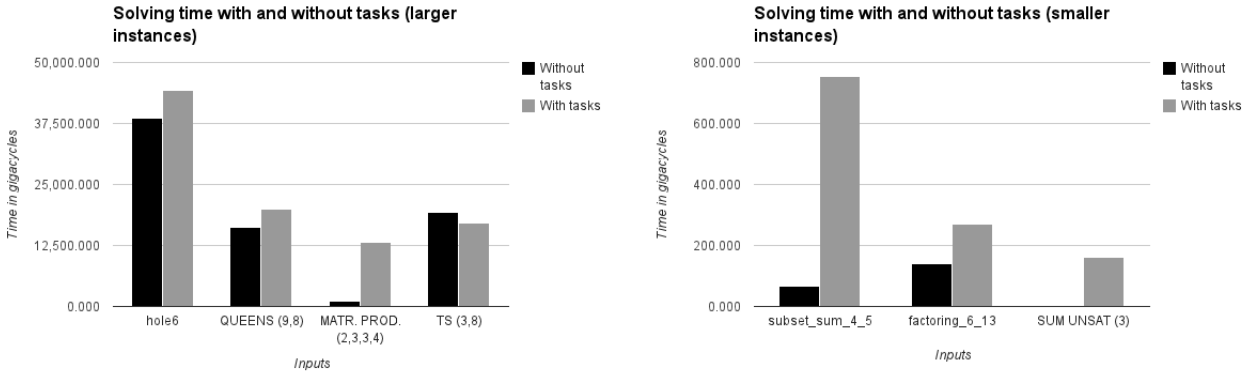


Figure 4.10: The sequential solving time for problems breaking them in tasks and running them at once.



We can easily see that the reason the time half the blocks was zero was because these blocks did not run any jobs, even though there were more jobs than the number of blocks. That brings some serious limitations, because if we are limit to 16 blocks, we cannot achieve the desired massive parallelism. Though we have not tested with a number of block between 16 and 32, this experiment indicates that 16 block are a limit of execution, since in an execution with 32 blocks, only 16 were actually doing some work. To further study this behavior is necessary to break this limit, by understanding this behavior.

4.2.5 Parallelization Overhead

Another challenge we have identified is the parallelization overhead. This overhead is associated with solving the problem with tasks. For a comparison, to test this overhead, we executed some problems sequentially (1 block and 1 thread), not using tasks and another time using 128 tasks. We did not include parallelism to avoid another variable to be included, but the execution with tasks can be parallelized by just adding blocks. Figure 4.10 shows the results. Both graphics show the same data, but since the scale was different for different inputs, we put the larger and the smaller inputs (in regard to solving time) in different graphs so it was easier to see.

After analyzing the results, we can see that the parallelization overhead is not too significant for most inputs and i t seems to be less important as the size of the input increases. To further explore this characteristic, we show in Table 4.1, the percentage of parallelization overhead. We calculated the different of the time with tasks and the time without them and calculated its percentage of the total time without task. The size parameter refers to whether it was in the first or in the second graph showed in Figure 4.10.

Of all larger instances, only one had a significantly large parallelization overhead

Table 4.1: Percentage Parallelization Overhead

Instance	***Overhead***	***Size***
hole6	14.76%	large
QUEENS (9,8)	23.66%	large
MATR. PROD. (2,3,3,4)	1009.16%	large
TS (3,8)	-10.96%	large
subset_sum_4_5	1039.93%	small
factoring_6_13	90.88%	small
SUM UNSAT (8)	18229.71%	small

and it happened to be exactly the smallest of them. The other ones all had some fairly small overhead, up to 24%, an overhead we expect to break by simply parallelizing the problem in two threads. The overhead percentage for "TS (3,8)" instance was negative, because it was even faster with tasks. Because it seems to decrease as the size increases, we believe this problem will be less important for larger instances. It is expected to behave this way, after all, the overhead is not dependent of the size of the input and is constant with a fixed number of jobs. This overhead covers a great portion of small instances, because compared to their execution, it is large. On the other hand, larger instances take such a larger time that the overhead is insignificant over all.

Furthermore, the assumptions that must be preprocessed due to parallelism are decisions that will not have to be made on the search. So part of the overhead actually excludes parts of the search. Because of that, we would expect it not to be too significant overall.

4.3 Improvement Proposals

This section presents two ideas meant to improve the behavior of SAT PaDdlinG. The first one is presented in Section 4.3.1 and has the objective of solving the workload distribution problem identified in some cases in SAT-PaDdlinG. The second proposal, in Section 4.3.2 has the main goal to allow SAT-PaDdlinG to actually take advantage of threads within warps, which was shown to be unfeasible in the current implementation.

4.3.1 Dynamic Search-Space Split

As we have shown in Section 4.2.2, the worst instances run with SAT PaDdlinG show low improvements in parallel due to bad workload distribution. As discussed in Section 2.4.2.1.1, a static division of the problem, as done by SAT-PaDdlinG fails as a

workload distribution of the problem, since the search-space is not uniform in average (SINGER, 2006), resulting in some very hard tasks to solve and some very easy ones, making some threads work more than others.

We propose the implementation of a dynamic split of the search-space, approach already used by many parallel SAT Solvers. In this approach, each running SAT Solver would start solving, as it does now. After some iterations (to be configured), the SAT Solver checks for idle threads, if it finds, it simply splits its current path, using the guiding paths approach and adds it to the list of tasks. To check for idle threads, a counter with global access, protected by atomic operations, would keep the count of idle threads. Whenever a thread becomes idle and does not find jobs to solve, it increments the counter. Whenever it verifies that more tasks are available, it choses one and decrements the counter.

The idle threads would need to keep polling the tasks list to find new jobs. Whenever they find a task, they solve it. They also need to check if every other thread is done solving, in such case, they must terminate their task. One way to accomplish that is to check whether the number of idle threads is equal to the number of total threads, a scenario that can only happen if all tasks are complete.

This polling strategy avoids unwanted threads communication and the threads remain independent. We still want to make sure that the low communication provided by SAT-PaDdlinG remains, specially because it was built to take advantage of massive parallelism. Along with this approach, a smarter algorithm for dividing the initial tasks is important. More uniform tasks would require less time for the threads to split their tasks and more time processing.

4.3.2 Second Level of Parallelism Proposal

Because of many limitations pointed out in previous sections, we have come up with an idea to cope most limitation, proposed in this section. Breaking the problem in many parts and solving each individual part seem to be a very obvious approach and its carried out by many SAT Solvers, including SAT-PaDdlinG. However, within the execution of a SAT Solver, we can try to parallelize part of the solution execution, by simple parallelizing the algorithms. For example, as presented in Section 3.2.2, in (FUJII; FUJIMOTO, 2012) it is presented an algorithm to automatically parallelize BCP.

Our proposal it to use only blocks to parallelize the input, by using guiding paths, as it is done now, and reserve the threads within the blocks to parallelize the algorithms, such as BCP. By parallelizing BCP, it will be most likely necessary to give up the use of two-watched literals, but it will allow parallel improvements, even in part of an already parallelized execution. The only problem with the solution used by (FUJII; FUJIMOTO,

2012) is that it requires a 3-CNF formula, which is not used by SAT-PaDdlinG and would have to be adapted.

BCP is not the only algorithm that could take advantage of that. To generalize, we propose to use the threads to parallelize the algorithms used by an instance of a SAT Solver. It would require more effort to come up with parallelization for the algorithms, but advantages would be found. And even only parallelizing BCP would already be a great improvement, since most of the process is done in BCP.

Since we would be within the same execution path, the SIMT problem would be diminished, we would break the 16 blocks limit, this parallelization would probably not require extra data structures, not bringing memory cost for parallelizing, it would use light-weight threads.

An important point is that such parallelization would not contradict to the current implementation of SAT-PaDdlinG. The current gains in parallelization would remain, since the instances themselves would be parallelized.

5 REPASAT

This chapter aims to study a new parallel environment in which a declarative and imperative code, with no regard to parallelism, is automatically parallelized by taking advantage of the SAT problem, the RePaSAT (Reduction to Parallel SAT) environment. Such an environment can take advantage of parallelization without the extra investment in parallel programming, resulting in shorter software development time and more reliable code. Since there are parallel SAT Solvers available, once a part of code is compiled into a CNF formula, it can be automatically solved in parallel.

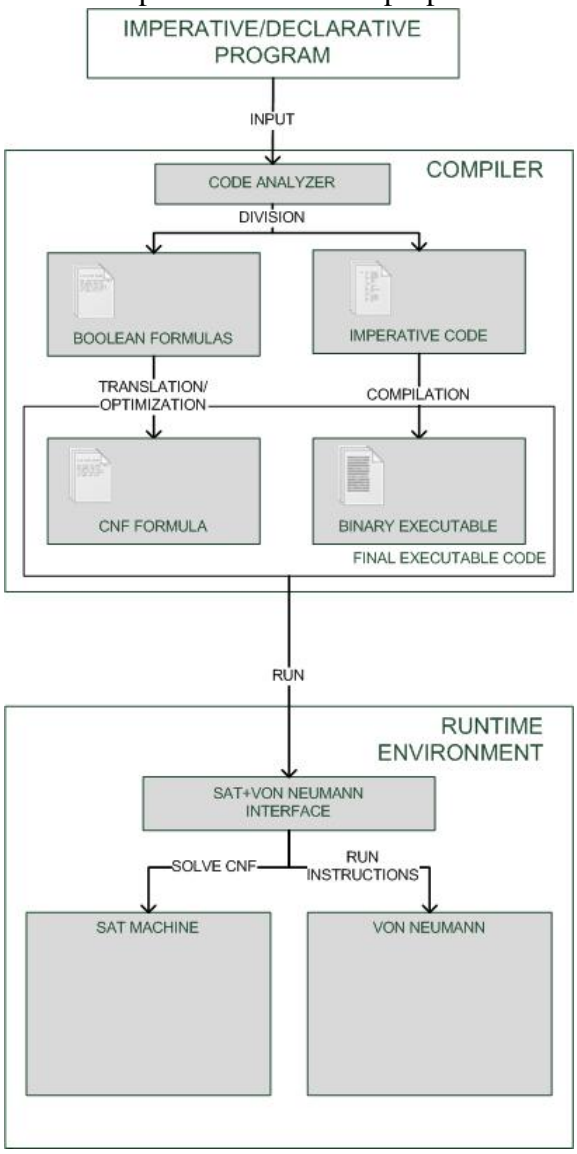
5.1 Concept

Given a declarative and imperative description of a problem, if this program can be translated as a CNF formula, it can be solved by a SAT Solver. Since there are parallel SAT Solvers, as shown in Chapter 3, the resolution of this formula can be automatically parallelized.

We can see in the structure of RePaSAT, depicted in Figure 5.1, that we generate two kinds of code to execute: CNF formulas and binary executables. It is depicted this way because it will not always be possible to represent the whole program as a Boolean formula, or even many Boolean formulas. That requires regular instructions execution, that must be run sequentially. Figure 5.1 does not imply that two executions (formula solving and instructions execution) are carried out in parallel. On the contrary, solving formulas become part of the code that is sequentially executed during the execution of the problem. We only show parts generated from the translation of code to this technique, that is, Figure 5.1 shows the structure not the behavior.

In RePaSAT, any given imperative and declarative code is analyzed by a code analyzer that determines the parallelizable part of this code. After studies we have been unable to turn some parts of codes in CNF formulas, such as loops whose number of iterations were not determined at compilation time. Thus, the code is actually broken into parallelizable parts (that can be turned into formula). From then on, we have two kinds of

Figure 5.1: Representation of the proposed RePaSAT



execution, that are sequentially run, only the formulas are actually parallelized.

In our approach, the formulas are generated once and they become part of the code solving them. There is also the idea of generating the formulas on-the-fly. In other words, rather than compiling code into formulas and then executing the remaining code and solving formulas, we could keep translating the code to formula and running it in run-time. However, since we want to avoid an extra overhead of translating formulas to code in run-time, after all, our objective is to improve the efficiency of the code, in our approach we always compile the code into formulas once and then we run the code + formulas resulting code.

An interface to this machine receives both the CNF parallelizable formulas and imperative code. In this code, we have a 'call' for solving formulas. They are both run by this machine, though the formulas are parallelized, as opposed to a simple run of the input code, which would be completely sequential.

The most important advantage RePaSAT provides is the possibility to parallelize even code with a large chain of dependencies. For example, imagine a sequence of operations in which every value is dependent of its previous. Algorithm 2 shows an example.

Algorithm 2: An algorithm with a chain of dependencies

```
input a;
b := a + 2;
c := b - a;
d := c - 1;
e := d * d;
```

This sequence of code is intrinsically sequential, because every line of code is dependent on its previous. However, when expressed as a formula, this chain of dependencies is undone. In a formula, there is no sequence of operations, only a search for a solution, which can be done in parallel. This may allow the parallelization of previously 'unparallelizable' code.

5.2 Code to Formula Translation

The whole idea behind RePaSAT is predicated on the possibility of translating code into a CNF formula. This can be accomplished because data types used by program can be represented as Boolean variables. These types, when stored in memory, even in regular imperative languages, are represented as a set of bits, a binary representation. In the formula, the operations carried out on these data are represented as constraints, as clauses in a SAT formula, that map a set of bits, the input, to another set of bits, the output. Ursa is already capable of translating imperative operations (arithmetic and Boolean) into

clauses (JANICIC, 2010).

When we mention declarative code, we mean code expressed as Boolean expressions, rather than instructions to be followed. Since the SAT problem solves a declarative description of a problem, with constraints modeling the problem, rather than a set of instructions to solve it, translation from a declarative specification to a SAT formula is rather trivial, so we added declarative code as an extra feature to RePaSAT. Furthermore, Ursa (JANICIC, 2010), the tool we used for code reduction to SAT is already capable of translating from a declarative and imperative code.

When starting with a specification in imperative code, it is first necessary to identify those parts of the code that are parallelizable. One simple approach would be to identify as much code as possible, stopping at any code that we are unable to parallelize. For example, consider the Algorithm 3.

Algorithm 3: Example of code to be parallelized in imperative description.

```
input a ;
b := a * 7;
for 1 .. 20 do
    b += 40;
end
c := a ;
while c < 20 do
    c += b;
end
```

Notice that the first loop is not data-dependent and can be entirely transformed into a formula. The second loop, however, has a condition that does not allow one to know before hand how many iterations it will have. As described in Section 5.2.3.3, the first one can be translated to a formula, while the second cannot. This way, everything before the second loop can be translated as a formula, we will call it f_1, as well as everything inside this loop to another formula, f_2. We can translate this code represented in Algorithm 4.

Algorithm 4: Code modified to allow parallelism.

```
input a;
solve(f1, {a}, {b, c}) ;
while c < 20 do
    solve(f2, {b}, {c});
end
```

The Algorithm 4 shows the code modified for parallelization. We have an operation, solve(f, I, O), which receives a previously generated formula, f, that executes parts of the code. It receives a set of inputs I, which are values that must be mapped to

variables in the formula and a set of outputs O that represents the values after processing the formula. Since the formula is run by a parallel SAT solver, this part of the code is completely parallelized. The individual steps remaining in the code, however, must be run sequentially, since they cannot be represented as a formula. That is, we first solve f_1, in parallel, then we execute that while loop. For each iteration, the loop solves formula f_2 and terminates once the input c is larger or equal to 20.

The formulas f_1 and f_2 are generated and stored along with the code, as depicted in Figure 5.1. It is also necessary to store which bits represent their inputs and which bits represent their inputs. Allowing inputs and outputs for formulas is described in Section 5.2.2.

5.2.1 Ursa

There is already available a tool capable of translating imperative/declarative code into a SAT formula. Ursa (JANICIC, 2010) receives a specification using its own imperative/declarative code language and generates formulas to solve a problem. Ursa either runs the operations imperatively or symbolically and only in this second case we can generate formulas for the inputs. Ursa has the "assert" instruction, which receives a Boolean expression. This Boolean expression may use variables which values are defined by Boolean constraits. Whenever Ursa reads this instruction, it generates a SAT formula describing the Boolean operation and solves it with a SAT Solver.

Although Ursa can represent many operations as a formula, it is still limited. Ursa was not created to parallelize code, rather, it was created to solve problems with a SAT Solver. For that reason, many operations are not solved as Ursa reads the file. In our approach, this would make these operations not parallelizable, that is why we study ways to actually represent these operations in the formula, presented in the following sections.

5.2.2 Formula Inputs and Outputs

As we have shown in Algorithm 4, the solution of the generated formula must accept inputs and outputs, since we want to generate it once and use it in every execution. Regardless of what types the inputs or outputs they are, in the formula they will be represented as a set of bits. For instance, an integer number can be represented as 32 variables (bits) in the formula.

The formula is modeled as to accept any value to the inputs, but to constrain the outputs based on the inputs. If we want to solve a formula, given an input, this input represents a set of variables in the formula that we want to set. For example, if we want to set an integer variable described by 4 bits as the number 10, we can represent as 1010_2. That is, we will have 4 variables in the formula, two of them we will set as true (1) and 2

of them as false (0).

To accomplish that, we can either use unary clause, which are clauses in the formula formed by 1 literal, or use assumptions, for SAT Solvers that support them. Unary clauses are useful, because we can simple append them to the rest of the formula and solve it as a whole. That is, the formula is generated in compilation time and the input is just appended afterward. Assumptions allows us to used completely unmodified formulas.

The outputs are only mappings from variables in the formula to bits in the code. Once the formula is solved, the bits that represent the formula can be mapped back to the variables representing the output and the execution can proceed.

5.2.3 Translation of Operations

Given a set of operations to be executed, all we want to do is to represent them all in a formula. To accomplish that, we need to represent them as a set of constraints that, given an input, make the output hold the expected result. Given that a single operation can be express as a formula, the chain of operations can be expressed as a conjunction of those formulas.

For example, imagine we had three variables a, b and c, which we would represent as a set of bits in the formula. We have operation: $a := b + c$. There is a set of Boolean operations for the bits in b and c that represent their sum, as shown in Section 5.2.3.1. Thus, all we have to do is make a be equal to this sum, which can also be expressed by Boolean operations. This entire operation can be expressed as a formula.

Imagine we had a set of operations, since all their values must be true, we can connect them by conjunctions. This way, we have represented the entire code as a single formula and solve it in parallel. However, in a formula, a variable will only hold a single value. If in the code we have variables that hold more than one value along the course, we need to represent each of these values as different variables in the formula. That is, if the same value appears more than once in the left side of the attribution, it must be represented with more than one variable in the end.

To represent any operation as a CNF formula, all we need is to represent it as a Boolean expression. Since any Boolean expression can be translated to a CNF formula in linear time (KROENING; STRICHMAN, 2008), this expression can be solved by a SAT Solver. Since many common operations, such as arithmetic and Boolean are already translatable to CNF by Ursa, many parts of code can take advantage of this technique. Also, we can express equality, as Ursa does, forcing a variable to be equal to an expression, allowing us to express attribution.

5.2.3.1 Arithmetic and Boolean Operations

Ursa is already capable of translating arithmetic operations into Boolean operations to be represented in the formula (JANICIC, 2010). Given two numbers represented as vectors of bits (refer to Section 5.2.3.5), we can represent arithmetic operations over those bits as Boolean operations. Ursa is capable of represent these operations over integers: sum, subtraction and multiplication (JANICIC, 2010).

Though it does not have embedded the integer division operation, in (JANICIC, 2010) it is shown an easy way to represent it, by using the multiplication operation. If given the variables a, b and c, we want to represent $a := b/c$, where '/' represent the integer division, we can simply express it as $b == a \times c + r$ (JANICIC, 2010), where r is the remainder of the division, with $r < a$ and $c < b$ (JANICIC, 2010). The interesting point about this approach, is that since we are describing this operations as constraints, it does not matter if the value we want is before or after the the equality, if we set the values b and c, for example, with unary clauses, the value a is forced to have a value that satisfies this expression and it can be the output.

To translate Boolean expression, since the formula will represent Boolean operations in the end, all we have to do is to map every bit into the expected Boolean operation. That allows us to represent many bitwise operations. Operations with real numbers, using either fixed-point or floating-point are not supported by Ursa. In Section 5.2.3.5, we present our approach to represent fixed-point numbers and operations.

5.2.3.2 Conditional Operations

In Ursa, conditional commands can only involve ground values (JANICIC, 2010), that is, numbers that are already set before generating the formula. That is, because Ursa does not represent conditionals as part of a formula, but execute them when reading the file. However, if we do not express conditions in the formula, this will be an unwanted restriction, we will not be able to parallelize code with conditionals. To accomplish that, we have come up with a Boolean representation which is equivalent to a conditional operation.

Algorithm 5: A conditional operation.

```
if condition then
  | operation1;
else
  | operation2;
end
```

Given any algorithm as in Algorithm 5, considering that condition is already a Boolean operation and the operation can be represented as a Boolean operation, we can

represent this code as represented in Equation 5.1. The function $to\_bool(ops)$ translates the imperative operations 'ops' into Boolean expressions.

$$(condition \wedge to\_bool(operation1)) \vee (\neg condition \wedge to\_bool(operation2)) \quad (5.1)$$

Given a set of inputs that force the condition to be either true or false, the formula works out as to make either the operation1 or operation2 to be true, given the conditions. It is also possible to allow conditions without the 'else' part, only removing the 'operation2' from the formula. It is necessary, though, to keep the negated condition, to allow the 'operation1' not to be executed. Notice that the Equation 5.1 is not in CNF and must be translated.

5.2.3.3 Loop Operations

In our approach formulas are immutable, except for adding input values (see Section 5.2.2). RePaSAT is incapable of representing loops without knowing the number of iterations it takes when actually generating the formula. Given any loop, it is necessary to unroll it so we have each of its iterations represented in the formula.

Loops whose number of iterations is unknown in compilation time, must be executed imperatively. It is possible, though, to represent the inside of the loop (given no loops within) and solve the formula many times. To explain further, in the first compilation step of RePaSAT, we must generate formulas. If the number of iterations of a loop is dependent (directly or indirectly) from the input, we cannot know how many iterations we will have to represent within a formula, as the iterations have to be carried out in run-time.

On the other hand, if we can determine the number of operations, in the case it is based on a constant value, we can simply unroll the iterations and express them all as a formula. In this case, the formula will grow as the number of iterations grow, but we will be able to parallelize all iterations, which is a good advantage.

5.2.3.4 Structure Representation

To parallelize any code using a CNF formula, we must be able to represent all data structures in it. Since CNF formulas only hold variables, this can be a hard task for many structures, specially those that represent dynamically changing structures. Also, we need to represent the intermediary results for the variables, in case they change their values. In a dynamic structure, if it is too large and has many modifications, we may need to have many different versions of it, making the solution unfeasible.

As far as this research goes, we have only expressed algorithm that use only these four types of structures: Boolean type, numeric type, vector (of Boolean or numeric) or matrices (of Boolean or numeric). Notice that these structures can be used to represent others, for example, in some cases we use matrices to represent graphs.

5.2.3.5 Number Representation

We studied two kinds of representation of numbers: integer and fixed-point. As Ursa represents the number as a vector of Boolean variables, so did we. However, Ursa does not give any kind of support to fixed-point numbers representation or operations. These can also be expressed as a vector of Boolean variables. Thus, we used Ursa numeric representation to hold fixed-point values and we implemented fixed-point operations.

Integer numbers. Integer numbers are represented as a vector of Boolean variables. Since Ursa's operations were used, these only have support for positive values, though negative values can be implemented through two's complement representation.

Fixed-point numbers. Fixed-point numbers are also represented as a vector of Boolean variables. Given N bits used to represent the number, I are separated to represent the integer part, the most significant part of the number, and D is separated to represent the decimal places, where $N = I + D$. These values are fixed for an execution.

The fixed-point operations we have implemented are: sum, subtraction, multiplication and division. To better understand how we implemented the number, think that integers and fixed-point numbers are sets of bits. So, any fixed-point number (with $D > 0$) can be temporarily interpreted as an integer number (with $N = I$ and $D = 0$), by placing the point as at the right end of the number, just so we can apply already existent integer operations on them. Whenever we interpret a number as integer, we are actually shifting right D places.

For example, for $D = 4$ and $I = 4$, the number 1.25 is represented in bits as depicted in Equation 5.2. Since we have 4 decimal places, if we ignore the point we will have the number 20, shifting 4 places to the left or multiplying by 2^D, with $D = 4$. In other words, for any fixed-point number f, if we interpret it as an integer i, we will have the equivalent number $i = f \times 2^D$, where D is the number of decimal when it was a fixed-point number. That gives us the condition in Equation 5.4, where « is the shift left operation, f is a floating point number and i its integer counterpart. In the same way, whenever we apply an integer operation op_i, it is equivalent to its fixed-point counterpart operation (op_f) divided by 2^D, where » is the shift right operation, since the result is an integer that must the converted to floating point, as shown in Equation 5.5.

$$1.25_{10} = 0001.0100_2 \tag{5.2}$$

$$20_{10} = 00010100_2 \tag{5.3}$$

$$i_n = f_n \times 2^D \Leftrightarrow f_n \text{«} D \tag{5.4}$$

$$a\ op_i\ b = \frac{a\ op_f\ b}{2^D} \Leftrightarrow (a\ op_f\ b) \text{»} D \tag{5.5}$$

Since we have integer operations available and we want to use them to operate floating point operations, we present in Equations 5.6, 5.7, 5.8 and 5.9 equivalences of those operations. We denote $+_i$, $-_i$, $\times_i$ and $\div_i$ as the integer sum, subtraction, multiplication and division operations already available. We denote $+_f$, $-_f$, $\times_f$ and $\div_f$, their fixed-point counterparts that we want to implement using the previously defined ones.

$$i_1 +_i i_2 = f_1 \times 2^D +_i f_2 \times 2^D = \frac{f_1 \times 2^D +_f f_2 \times 2^D}{2^D} = \frac{(f_1 +_f f_2) \times 2^D}{2^D} = f_1 +_f f_2 \tag{5.6}$$

$$i_1 -_i i_2 = f_1 \times 2^D -_i f_2 \times 2^D = \frac{f_1 \times 2^D -_f f_2 \times 2^D}{2^D} = \frac{(f_1 -_f f_2) \times 2^D}{2^D} = f_1 -_f f_2 \tag{5.7}$$

$$i_1 \times_i i_2 = f_1 \times 2^D \times_i f_2 \times 2^D = \frac{(f_1 \times_f f_2) \times 2^{2D}}{2^D} = f_1 \times_f f_2 \times 2^D \tag{5.8}$$

$$i_1 \div_i i_2 = (f_1 \times 2^D) \div_i (f_2 \times 2^D) = \frac{(f_1 \div_f f_2)}{2^D} = \frac{f_1 \div_f f_2}{2^D} \tag{5.9}$$

Equations 5.6 and 5.7 show us that the integer sum and subtraction of the integer representation of two floating-point numbers result in the sum or subtraction of the floating-point numbers themselves. For that reason, we can get these number and use the integer sum or subtraction, already represented in a SAT Solver and interpret the result as a fixed-point number.

Multiplication and division, on the other hand, require different strategies. The Equation 5.8 and 5.9 show us that if we use integer multiplication and division on two fixed-point numbers, we end up with a number 2^D times larger in the case of multiplication and 2^D smaller in case of division. To solve the multiplication problem, we can use the integer multiplication and shift the result D places to the right.

However, this may bring some precision problems. For example, with $I = 4$ and $D = 4$, when multiplying 12 and 1.25, we would have 15 as result, all fit our precision of 4 integer places. However, during the operation, we would have the intermediate value of 240, which cannot be represented, for it needs 8 integer bits. One possibility for that is to shift right half of D for each operand and simply multiply. In other words, we use $f_1 \times f_2 = (i_1 \gg \frac{D}{2}) \times (i_2 \gg \frac{D}{2})$. This way, we may lose some decimal bits of the operands, but do not run the risk of not having precision enough to represent the solution. This is the approach we chose.

We can also add signs to the multiplication, using two-complement. We are forced to test 4 different possibilities or operands: 2 positive operands, 2 negative operands, first positive and second negative and first negative and second positive. In the first 2 configurations, we have an output that is positive, but in the second case we must undo the two-complement. In the last two possibilities, we undo the two-complement in the negative number and redo it after multiplying, for the result is negative.

As for the division, we can use the approach Ursa shown in Section 5.2.3.1 and adapt it to fixed-point operations. For two numbers, f_1 and f_2, if we want their division, that we call f_3, considering that there will be a remainder r, we express the division as the Boolean expression in Equation 5.10: This will bring us the ratio of the two numbers, such that the precision grows with the number of decimal places. If by any chance f_2 is not an integer, by shifting it D places to the right, we will be rounding it to the previous integer. Thus, for instance, dividing 1.25 by 2.25, will be equivalent to dividing 1.25 by 2.

$$f_3 = \frac{f_1}{f_2} \Leftrightarrow f_1 == f_3 * (f_2 \ll D) + r \wedge r < f_2 \wedge f_3 < f_1 \tag{5.10}$$

By shifting right the divisor (f_2) in D decimal places, we are actually dividing it by 2^D or multiplying the result by it, which compensate the problem specified in Equation 5.9. We could also shift left the dividend in D places with the same result, without losing the precision of the divisor. However, we chose the first approach, because the second may cause the SAT Solver to shift the dividend more places than the precision, losing significant bits, instead of non-significant ones (decimal places).

To analyze the precision with this technique with the multiplication and division operations, which are the ones that are not a simple application of integer operations, we have run an experiment in a set of numbers. First we calculated the multiplication and division of the number in a regular calculator. Then, we represented the number in our fixed-point representation, with $D = I = 16, N = 32$, we solved it in Ursa, with the SAT Solver Clasp. Since Ursa always interpret the numbers as integer and displays it this way, we divided the result by 2^{16} with the same calculator to get the results as a real

Table 5.1: Comparison between the results of multiplication and division in a calculator and solved by a SAT Solver

Input		Multiplication			Division		
Operator 1	Operator 2	Calculator	Clasp	Difference	Calculator	Clasp	Difference
23.4	12.2	285.48	285.443	0.037	1.918	1.950	0.032
43.5	54	2349	2349	0	0.806	0.806	0
7.987	23.918	191.033	190.970	0.063	0.334	0.347	0.013
15	18	270	270	0	0.833	0.833	0
123.43	12.343	1523.496	1523.103	0.393	10	10.286	0.286
202.32	202.32	40933.382	40931.928	1.454	1	1.002	0.002
155	155	24025	24025	0	1	1	0

number. All numbers were rounded to the third decimal place. The absolute difference (the positive difference) is shown to see the error caused by calculating these operations with a SAT Solver.

We can see in Table 5.1 that these operations with a SAT Solver are fairly precise, showing results very close to a common calculator. The largest difference, which was the only one greater than 1, was 1.454. Unexpectedly, it was for a multiplication operation, rather than a division which would be expected, since divisions always round the divisor to the previous integer. As for the others, in decimal, there are only 2 cases where the results where precise to less than 1 decimal place, and 6 cases where the difference was calculated as 0 (close enough to be zero with 3 decimal places of precision). We conclude that this approach can be fairly close and, if greater precision is necessary, more decimal places can be added.

5.2.3.6 Vectors and Matrices

To represent vectors and matrices, we have used the Ursa approach, basically representing a set of numbers. To represent a vertex, since its size must be fixed in compilation time, a limitation for this approach, we can represent it as a simple set of many basic variables. Matrices work in the same way, only we have two dimensions, as though we had a set of vectors. Other structures with more dimensions could be represented using this approach.

5.3 Parallelization and Implementation

The automatic parallelization guaranteed by RePaSAT stems from parallelization of SAT Solvers, as presented in Section 2.4. If an entire program is turned into a SAT

formula, it can be automatically parallelized, taking advantage of the parallelism brought by parallel SAT Solvers.

However, it seems that not every specification in common imperative can be expressed as Boolean formula. Therefore, it is necessary to break the code into parts that can be turned into formulas. Only those formulas can be parallelized, allowing us to parallelize portions of code that overall are sequentially executed. Thus, we need to generate formulas that are large enough to compensate the parallelization, and not too large to render the solution unfeasible.

One way to implement this parallelism is to 'plug' a parallel SAT Solver to Ursa. Ursa is already capable of using different SAT Solvers. It actually requires their code, since the code is embedded with its code and it requires modification of Ursa's code. This approach can makes immediately any code developed in Ursa parallelizable.

However, since Ursa was not build with this objective, rather, it is proposed as a modeling system (JANICIC, 2010), to allow programming with imperative and declarative commands. Also, Ursa is interpreted (JANICIC, 2010), it reads the file as it executes it, without compilation, and generates the formulas during runtime. This does not comprise with our proposal, to generate formulas for parts of the code and replace them with a call for solving those formulas.

As far as this research go, we only intend to study two different characteristics of this machine, how parallelizable the generated formulas are, to shed some light on the parallelization capacity; and to analyze the overhead it brings, as present in Section 5.4. With that in mind, we have projected prototypes of this machine. For experiments in Sections 5.4.1 and 5.4.2 and the Traveling Salesman Problem presented in Section 5.4.3, we implemented the entire code in Ursa, using the techniques presented so far. The Ford-Fulkerson Algorithm used in the experiment in Section 5.4.3 required the aid of bash script, because Ursa is not capable of reading results from formulas, only displaying them. Because we could not express the entire problem as a formula, because is a loop whose number of iterations is not determined before running and we need outputs from the formulas, we need aid from bash.

It is important to point that Ursa allows the developer to configure the number of bits to represent integer numbers. It only provides two types, Boolean and integers, Booleans are always represented with 1 bit, while the integers are configurable. Ursa does not allow the use of different number of bits per variable. The number used to represent a variable is set in the beginning and every variable will have this configuration. In our experiments, when we mention the number of bits used in Ursa, that is the number we refer.

5.4 Preliminary Results

We have carried out experiments to analyze parallelization through SAT reduction and the basic behavior of this technique. We used Ursa (JANICIC, 2010) to represent code as CNF formulas, allowing us to automatically parallelize their code. We presented in Section 5.3 a description of how the formulas presented here were generated. We present three sets of experiments. Two speedup experiment using the time to solve generated formula, not always the algorithm or problem as a whole, since we intend to test the parallelism of the formula. These are presented in Sections 5.4.1 and 5.4.2.

The objective of these experiments is to check the first step towards the expected automatic parallelism provided by RePaSAT: checking if the formulas are parallelizable. If we do not manage to parallelize the formulas, there is no hope to take any kind of advantage from RePaSAT, since its contributions lie on parallelizable formulas.

Another experiment is shown in Section 5.4.3, where we intend to present the overhead brought by using SAT to execute code. We compare code generated in C language against code generated in Ursa. Thought the implementations are not exactly the same, we intend to analyze if the NP-Completeness of the problem will make the execution too hard to improve with parallelism.

From the execution of some problems, modeled either in Ursa or an Ursa + bash scheme, we have come up with some executable code that successfully solves some problems. These problems generate one or more CNF formulas to solve the problem. These formulas may or may not include the entire problem, but the problems are always fully solved, even if processing beyond the SAT solving is necessary. The inputs we used are listed below:

- **SUM UNSAT** : An example to show the capability of declarative code. This code attempts to find two positive integer numbers whose sum equals 4 and subtraction equals 5. No such two values exist, so this problem is unsatisfiable. This entire problem is represented as a single formula.
- **QUEENS** : Another declarative code, this solves the problem of placing N non-attacking queens on a NxN chess board. This problem is solved as a single formula.
- **TS** This is an declarative code that solves the Traveling Salesman problem with N cities. This problem is solved with a single formula.
- **MATR. PROD.** : This is an imperative code that solves the problem of multiplying one matrix PxQ and another QxR. This problem generated a single formula.
- **GRAPH REACH.** : Determines whether, in a graph with V vertices and E edges, there is a path that connects one chosen vertex to another. A matrix was used to

Table 5.2: Inputs for experiment with speedup with SAT-PaDdlinG

Input		Resulting formula		Ursa Config.
Algorithm	Input Size	# of Variables	# of Clauses	# of Bits
SUM UNSAT	-	23	66	3
QUEENS	$N = 9$	225	1110	8
TS	$N = 3$	375	1450	8
MATR. PROD.T	$P = Q = 2$ and $R = 3$	750	2544	4
GRAPH. REACH.	$V = 11$ and $E = 12$	323	3166	3
LCS	$N = M = 10$	1861	1150	8
BUBBLE SORT	$N = 5$	1421	3086	8

represent the graph and the path that connects them (if it exists) is returned. This was created with a declarative code that generated a single formula.

- **LCS** : Solves the Longest Common Subsequence problem, for two strings of size M and N. An imperative and declarative code was used that generated many formulas. The operation "maximize", provided by Ursa, was used, that allows us to run many formulas and set a value to a variable such that it has the largest value within a range that satisfies the constraints that describe the formula.

- **BUBBLE SORT** : This is the Bubble Sort algorithm run for a vector of size N. An imperative code was used, which generated a single formula.

The instances name are followed by their input in parentheses. This is a list of their inputs and the last element in the number of bits set in Ursa to define numbers.

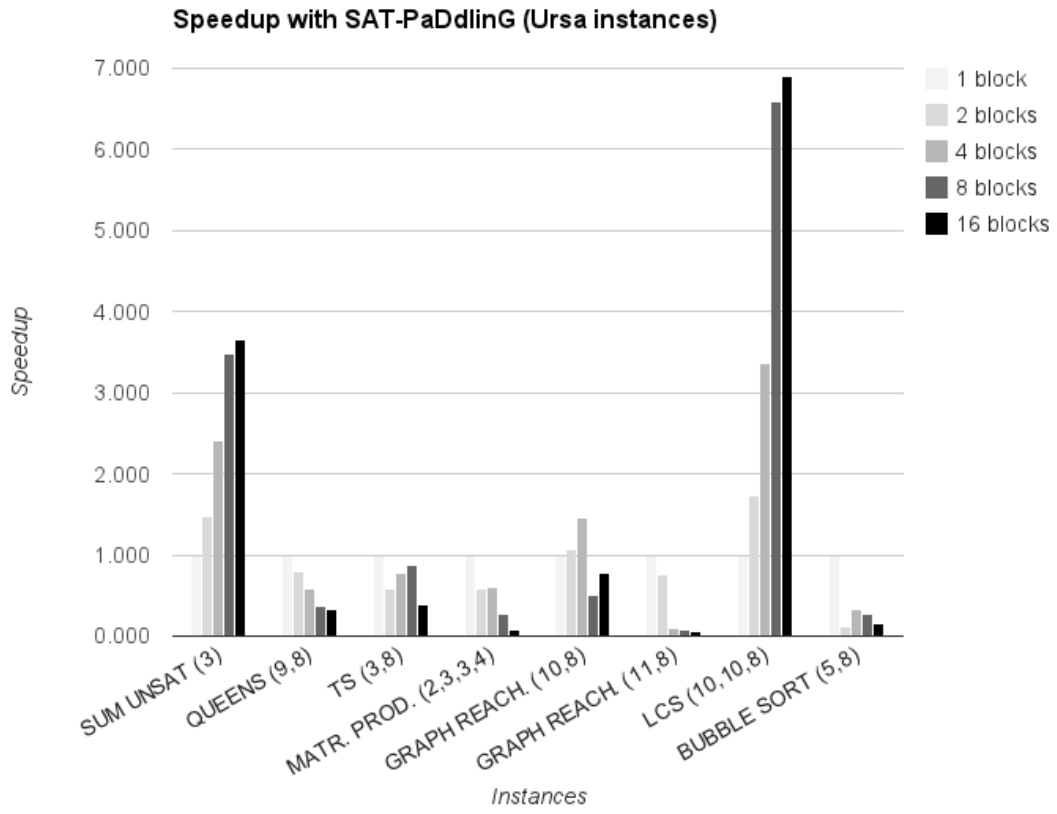
5.4.1 Speedup with SAT-PaDdlinG

SAT-PaDdlinG is our proposed parallel SAT Solver presented in Chapter 4. We have executed the input instances and tested a sequential run in SAT-PaDdlinG against parallel run with 2, 4, 8 and 16 blocks and have taken the speedups. The sizes for inputs for this experiment are shown in Table 5.2. It is important to point out that some inputs generated more than one formula, only one is shown.

This experiment was run in a Intel Core i7, with 2.3GHz and 8GB of RAM memory. The GPU on which the SAT Solver was run was a NVidia GeForce GT 630M, with 96 CUDA cores, 800MHz of processing speed and 2GB of GPU memory.

Figure 5.2 shows the results. We can clearly see much slower speedup when comparing to the ideal speedup, for most instances. Speedup smaller than 1 indicates that the problem has become slower in parallel, which is contrary the purpose of parallelizing in

Figure 5.2: Speedup for instances generated with Ursa and run in SAT-PaDdlinG.



the first place. For every instance, the execution with 1 block has always 1 of speedup, because it is the version for comparison. Apart from “SUM UNSAT” and “LCS”, every instance had most speedup slower than 1, becoming slower in parallel. From those instances, only “GRAPH REACH.” showed speedup greater than 1, but only for 2 and 4 blocks and only reaching 1.45 of speedup for 4 blocks, still very close to 1.

The instances “SUM UNSAT” and “LCS” show significant speedup. Particularly “LCS”, which show interesting increase, reaching a speedup of 6.9 for 16 bits. However, if in 8 examples, only 2 can significant advantage of parallelism, this indicates that this approach is not one that provide parallelism to compensate its usage. As a matter of fact, we show in Section 4.2 better results for inputs that are not generated from Ursa. Also, in Section 5.4.2 we show that ManySAT shows a similar results. This seems to be a characteristic of the formulas generated by Ursa.

5.4.2 Speedup with ManySAT

We present here some speedup results using ManySAT. To make sure the results in Section 5.4.1 were not a limitation of SAT-PaDdlinG, GPU or a cooperative approach to these generated instances from Ursa. ManySAT, presented in Section 3.1.1, in a CPU SAT Solver that uses the portfolio approach, so it parallelizes the problem in a completely different way from SAT-PaDdlinG, if also this tool is incapable of parallelizing the formulas, we will have a strong indicative that the generated formulas provide insignificant improvements in parallel, either for the way of how they are generated by Ursa, or because the problems themselves are not parallelizable, even by SAT.

Because ManySAT is built over MiniSAT and has improvements that we have not been able to add to SAT-PaDdlinG yet, ManySAT is capable of solving larger instances than SAT-PaDdlinG more quickly. Also, when solving the formulas used in experiment for SAT-PaDdlinG in Section 5.4.1, ManySAT solves them so fast and the tool we used to measure time return 0 is most cases. For that reason, we used different sizes, to allow larger formulas to be created. Though the formulas are different, they represent solutions for the same problems, with the same characteristics. ManySAT only allows 4 threads, so we tested with all possible configurations of threads: 1, 2, 3 and 4, to have more data do analyze. The sizes used in this experiment are presented in Table 5.3.

This experiment was run in an Intel Core i5-2400 CPU with 3.10GHz and 8GB of RAM with an Ubuntu 11.10. This machine includes 4 cores, which are used from parallelism, as opposed to GPU.

We can see the results in Figure 5.3. Since speedup lower than 1 indicates that

Table 5.3: Inputs for experiment with speedup with ManySAT

Input		Resulting formula		Ursa Config.
Algorithm	Input Size	# of Variables	# of Clauses	# of Bits
SUM UNSAT	-	61424	163792	4096
QUEENS	$N = 30$	2640	34570	8
TS	$N = 11$	17620	144314	20
MATR. PROD.T	$P = Q = 2$ and $R = 3$	62406	212796	32
GRAPH. REACH.	$V = 25$ and $E = 26$	30050	838114	32
LCS	$N = 500$ and $M = 400$	340503	2100957	8
BUBBLE SORT	$N = 20$	96521	199526	8

Figure 5.3: Speedup for instances generated with Ursa and run in ManySAT.

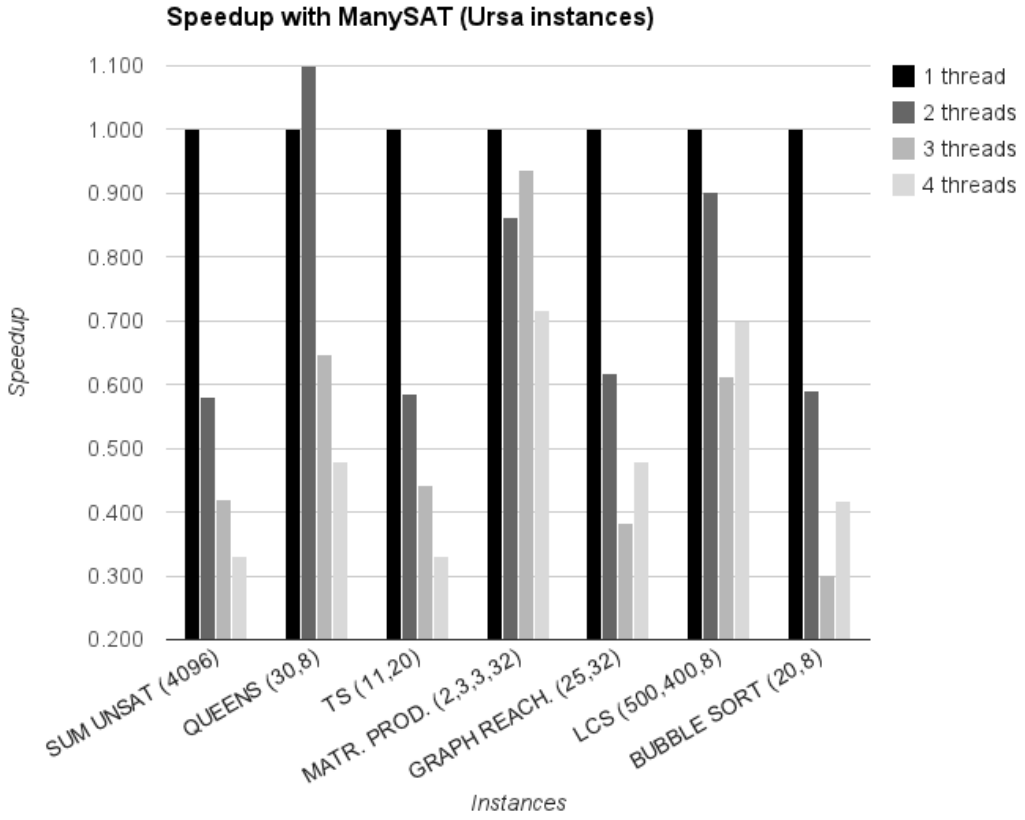
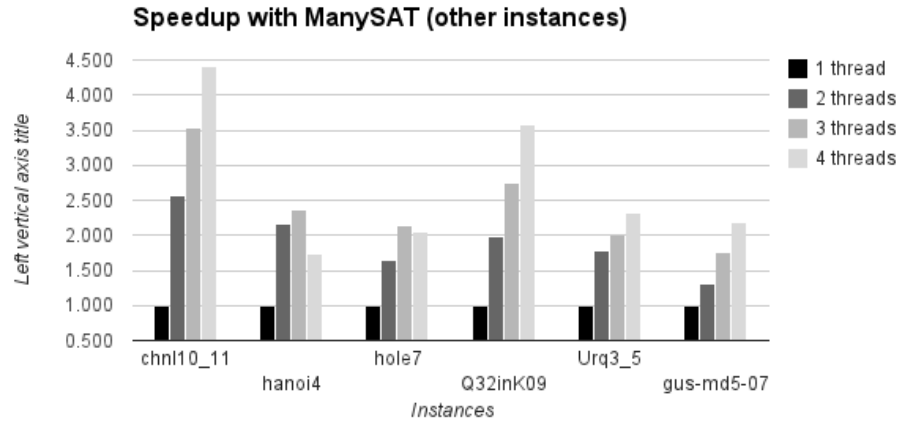


Figure 5.4: Speedup for instances from benchmarks, not generated by Ursa, and run in ManySAT.



parallel execution was worse than sequential one, we can see that the instances did not parallelize in ManySAT, like in SAT-PaDdlinG. There was gain in one single case, with the 'QUEENS' problem, but it only happened when using 2 threads, and the speedup was 1.1, though greater than 1, it is still insignificant and does not provide an expected parallelism.

The results with SAT-PaDdlinG and ManySAT indicate that the generated formula are not parallelizable. If we are unable to parallelize the formula, the only parallelizable part by using RePaSAT, the software as a whole will not improve as well. Experiments shown in Section 5.4.1 seem to indicate that SAT-PaDdlinG is particularly bad for instances generated in Ursa, with a speedup lower than 1, but in Section 4.2, it is shown significant speedup for other instances. If this behavior repeats for ManySAT, we can generalize that the problem lies on the generator of the formulas, the variable that was not modified. For that reason, we present in Figure 5.4 some results for instances not generated by Ursa, executed in ManySAT. All instances are from SAT benchmarks, they are: chnl10_11, hanoi4, hole7, Q32in09, Urq3_5, gus_md5-07.

One can clearly see in Figure 5.4 that ManySAT shows improvements in the parallel execution. All instances not generated from Ursa showed improvements in parallel execution, most showed significant improvements. We can see superlinear speedup, which is a speedup greater than the ideal one, which is a speedup equal to the number of threads. The instance 'chnl10_11' showed superlinear speedup in all configurations of threads, meaning it improved further than the ideal. The input 'hanoi4' had a superlinear speedup for 2 threads, but its speedup had a only a small increase for 3 threads and decreased for 4 threads, but was still better than sequential in all cases. The instances 'hole6', 'Urq3_5' and 'gus_md5_07' had similar behavior, they all had small speedup, reaching only a little more than 2, but still better than sequential. The instance 'Q32inK09' showed interesting speedup, more than 3.5 for 4 threads and superlinear for 2.

All instances in this experiment showed the same, ManySAT brings improvement in parallel, in some cases. The fact that all instances generated by Ursa were not parallelizable and all tested instances from benchmarks were parallelizable is a strong indicative that the formers are really not parallelizable because of how their generated. While this problem remains, it is impossible to take advantage of this technique.

A future approach is to try to determine whether there is some kind of input code that allows good parallelization. Our current results indicate that the Ursa generated code is not parallelizable, maybe there are specific kinds of code that are able to take advantage from this technique.

There is some evidence in the experiment in Section 4.2.2 that these problems are not parallelizable in SAT-PaDdlinG due to workload distribution problems. In fact, this is the only objective of that experiment. SAT PaDdlinG is yet to be improved when it comes to workload distribution and the experiments indicate that SAT-PaDdlinG may be take advantage of parallelism if this problem is coped with.

5.4.3 Technique Overhead

Because we have to change the code into an NP-Complete problem, even though we can break the problem into smaller formulas, we may expect some overhead coming from parallelizing via RePaSAT when compared to a sequential run. For that reason, in this section we show some experiments comparing a C implementation to verify if transforming code into an NP-Complete problem makes it much worse to solve.

To this experiment, we used two different inputs: an algorithm for the Traveling Salesman problem and the Ford-Fulkerson algorithm. As a SAT implementation, for the Ford-Fulkerson implementation, we have used the Graph Reachability problem used in Section 5.4.1, since this implementation not only determines if a path exist between two point of a graph, it also returns it. Since finding a path between two points in a graph is part of the Ford-Fulkerson algorithm, we used this part to be solved by a SAT Solver.

We have chosen these two algorithms because they are both run over graphs implemented in matrices. Since graphs are very common structures in many software and provide many variable connected one to the other, we believe this will provide an interesting case to study. It is important to point out that this section has the sole objective to estimate the overhead brought by this technique. For that reason, we ran these problems on a sequential SAT Solver, MiniSAT, and also had a sequential C version of the algorithms to test against. Because the approaches used to the implementation in RePaSAT and in C were different, we intend to simple identify very large discrepancies due to NP-Completeness of the problem. This experiment was run in the same hardware as experiment in Section 5.4.2. Results are shown in Table 5.4.

Table 5.4: Results for experiment with RePaSAT using MiniSAT and a C implementation

Algorithm	Input			Results (time)		Configuration
	#V	#E	Bits	MiniSAT	C	Repetitions
Ford-Fulkerson	80	156	7	56.334	0	10
Ford-Fulkerson	90	176	7	24.404	0	10
Ford-Fulkerson	100	196	9	170.178	0	10
Ford-Fulkerson	150	296	9	3921.601	0	10
Traveling Salesman	10	45	16	0.025	0	10
Traveling Salesman	12	66	16	184.835	4.619	10
Traveling Salesman	14	91	16	9.498	791.366	10

In this approach, we see an interesting behavior. The Ford-Fulkerson approach was so fast in the C implementation that is has '0' in all its cases. This means that it is not possible to calculate how many times it has become slower when solved by a SAT Solver. However, since the 'time' tool we used to get the running time is precise within a tenth of millisecond, the C code took less than 0.01 seconds to finish which means that the execution with Ford-Fulkerson for 150 vertices and 296 edges was at least 392160 faster in the C implementation. In other words, if we could perfectly parallelize it, we would need almost $400,000$ processing units just to compensate this overhead, considering ideal speedup.

The Traveling Salesman, on the other hand, had a strange behavior. We can see that it grows much faster in C implementation, which is expected, since the Traveling Salesman Problem is an NP-Complete problem. But we also see a very strong and unexpected improve from 12 to 14 vertices in RePaSAT execution. In this case, it becomes faster than the C implementation, becoming 81.3 times faster. We could never expect a problem become faster when its input size is increased. The best explanation for that, considering that it did increase from 10 to 12 vertices, is that the SAT Solvers' behaviors are usually difficult to predict. The generated formula possibly was by chance an easy to solve and that reflected the time. Of course, it may also have been the case that the execution with 12 vertices generated, by chance, a worse than expected formula and the one with 14 just got back to the expected difficulty.

From this experiment, we conclude that there may be problems in which the overhead of this technique is not killing, such as Traveling Salesman, which was better in one case, although it may have been to better suitable formulas. Even so, since we have been unable to parallelize the formula, as shown in Sections 5.4.1 and 5.4.2, this is irrelevant until this first problem is solved. Possibly, as the number of vertices increase, the SAT Solver become well suited for the Traveling Salesman Problem, since both are NP-Complete and it can take advantage of this approach.

The Ford-Fulkerson approach shows an expected characteristic: the sequential version run in SAT is worst than in C. This algorithm has worst time of $O(|E|C)$, where $|E|$ is the number of edges and C is the largest edge weight (KLEINBERG; TARDOS, 2006). Considering that C uses a constant number of bits to represent its numbers, it becomes a polynomial algorithm. Besides, it is an algorithm that has a loop that in which number of iterations cannot be determined in compilation time and hence only the inner part of the loop can be parallelized. These characteristics make it less suitable for a SAT Solver to solve it, explaining the results in this experiment.

This experiment shows that this approach is most probably unsuitable for some inputs, due to the large overhead it brings with it. Restrict it to NP-Complete problems may the answer or other domain identified as better suited. While it is not possible to parallelize the inputs with gains, it is not suitable as parallelization technique, but may serve as different approach to develop code.

6 CONCLUSIONS AND CONTRIBUTIONS

We have presented two different ideas that work together and have contributions on their own. In regard to them, SAT-PaDdlinG and RePaSAT, we have proposed the following topics for study:

1. The viability of parallelism of a DPLL SAT Solver on a SIMT environment, such as a CUDA GPU, the environment SAT-PaDdlinG is built in.

2. The implementation of SAT-PaDdlinG, as the option for massive parallelism for RePaSAT.

3. Some GPU characteristics preventing SAT-PaDdlinG to parallelize on this GPU environment, as well as proposals to cope with them.

4. The translation of imperative specification of programs as CNF Boolean formulas (translation necessary for RePaSAT).

5. The improvement of the generated formulas' execution time by means of parallelism.

6. The parallelization of these formulas on GPU environment (SAT-PaDdlinG). Analysis on a CPU for comparison.

For the item 1, we have arguably shown that we can take advantage from a GPU environment with a DPLL SAT Solver. We have reached high speedup in some cases, up to 16 blocks, which is more parallelism than found in other environments. We were able to have gains with parallelism even with workload problems, as presented.

However, the SIMT characteristic of GPU has shown to prevent the formulas from parallelizing and the use of different blocks is necessary to accomplish the mentioned gains. This is a way to prevent the SAT Solver from using the SIMT model, with the cost of having 31 unused threads per block. We have empirically shown that DPLL SAT Solvers cannot take advantage of a SIMT model by dividing the problem with guiding

paths. In fact, we were the fist ones to carry out experiments of a DPLL SAT Solver on a SIMT environment.

We have presented a proposal to add a second level of parallelism that may be capable of using this model in its advantage. By using parallelizing techniques already presented in literature, we may be able to take advantage of parallelism of procedures, rather than splitting the input. This proposal is complementary to the parallelism already employed by SAT-PaDdlinG and may be implemented alongside it, because the threads are independently solving the formula and their procedures may be independently parallelized, not affecting the current parallelism.

For the item 2, we have shown that it is possible to implement the most important and common optimizations for SAT Solver found in literature on CUDA. These optimizations included: the VSIDS decision heuristic, the two-watched literals scheme, with its data structures and algorithm, non-chronological backtracking, CDCL and restarts. We have presented our approach to these algorithms and have successfully solved formulas with them on GPU.

For the item 3, we have empirically shown that there are some issues to be solved to allow better parallelism for SAT-PaDdlinG. We have shown that SAT-PaDdlinG still has some workload distribution issues, it cannot parallelize by increasing the number of threads per block (the SIMT issue), only the number of blocks. It also reaches a limit of 16 block. In spite of that, we have reached significant speedup in some cases. We have been able to improve some formulas up to 16 blocks, which is usually better than what found on CPU, but it does not reach the massive parallelism we expected. We have presented two proposals to cope with these problems: a dynamic split of the problem, to code with the workload distribution problem and a second level of parallelism that will cope with the SIMT parallelization issue and the limit of blocks.

For the item 4, we have presented ways to express many different structures and operations in a CNF formulas. We have shown that Boolean, integer, fixed-point types may be expressed, as well as vectors and matrices. Arithmetic and Boolean operations over those types are provided. We have shown that conditional operations can be expressed, while loops with unresolved number of iterations cannot. We believe this includes a large number of applications and many codes may be translated. Parts of code can be chosen to be translated, if the entire code cannot.

For the item 5, we have shown our first challenge towards parallelization. We must be able to parallelize the formulas in better time than sequential to expect any gains from this parallelization approach. Our experiments show that the generated formulas have some distinctive characteristic that prevents them from parallelizing. This is the first thing to be coped with before parallelizing with RePaSAT. We also showed that this

technique shows high overhead in some cases and low in others, so domain reduction may be necessary.

As for the item 6, we have empirically shown that the generated formulas are not parallelizable in this GPU environment. We have also shown that other formulas are and the same behavior happens on a CPU SAT Solver. That strongly indicates that the formula has some characteristic that makes it hard to parallelize, as mentioned. A study of the structure of formulas and study of the generation of the formulas is advisable to understand this behavior.

Though we have been unable to parallelize code with RePaSAT, we have presented the first challenge towards it: generate parallelizable formulas. Since other formulas are parallelizable, that indicates that there may be ways to encode the formulas so they become parallelizable, unless this is a characteristic of the problem of representing code as formulas itself. We have provided different kinds of ways to represent many types and operations as formulas, to represent many different types of code.

During this research, as a GPU SAT Solver has become necessary, we have developed SAT-PaDdlinG. It not only allows RePaSAT to run on GPU, but also any other application that uses SAT Solvers. This SAT Solver was also used on the first empirical analysis of a DPLL SAT Solver on GPU and its limitations where exposed, but we were still capable of parallelizing it.

7 FUTURE WORK

We believe our two presented proposals can be greatly improved. As far as RePaSAT is concerned, as a continuation of our work, we believe that studying the generated formulas is the first step. Our experiments strongly indicate that is some characteristic of the generated formulas that stops them from parallelizing. Understanding this characteristic and how it affects the parallelization may be the key to generate parallelizable formulas, since there may be other ways to generate them. We believe there are two ways to approach this, either studying the generated formulas themselves or studying the generation carried out by Ursa. Also, understanding what makes formulas be parallelizable is important. We have empirical evidences that these formulas were not improving in parallel in SAT-PaDdlinG due to workload distribution problem. Possibly, the hiden characteristic in these formulas is that they are hard to distribute in different threads and strategies to cope with this characteristic may be the solution.

SAT-PaDdlinG already implements many algorithms that allows it to solve bigger formulas. However, there are still other algorithms available that would make it more competitive with other SAT Solvers, in particular the first-UIP for clause learning, regarded as a good learning algorithm and widely use, instead of the current implementation, which is from an old approach. A smarter clause deleting strategy would allow SAT PaDdlinG to hold more important clauses for longer, making it more efficient. Also, preprocessing on the formula before solving it would improve the process, for example, by removing pure literals. There are preprocessors that can be used, such as SatELite (EÉN; BIERE, 2005). Clause sharing will also help SAT-PaDdlinG, allowing the threads to cooperate. However, it must be implemented in such a way that it does not cause too much communication, compromising the independency of the threads.

Also, implementing the mentioned dynamic search-space split will help it to cope with its characteristic of bad workload distribution. The second level of parallelism is a proposal presented to allow SAT-PaDdlinG to take advantage of the SIMT environment it is inserted. By parallelizing two times the solution, by the input and the procedures, it should show much more speed up and provide a massive parallelism is this cheap envi-

ronment, without affecting the gains already presented, since the SAT Solvers objects are independent.

REFERENCES

AMBRUS, W. A framework for automatic parallelization of sequential programs. In: INTERNATIONAL CONFERENCE ON TELECOMMUNICATIONS, 2003. CONTEL 2003., 7. **Proceedings...** [S.l.: s.n.], 2003. p.693–696.

ARMSTRONG, B.; EIGENMANN, R. Application of Automatic Parallelization to Modern Challenges of Scientific Computing Industries. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 2008. **Proceedings...** IEEE, 2008. p.279–286.

BECKERS, S. et al. Parallel hybrid SAT solving using OpenCL. In: BENELUX CONFERENCE ON ARTIFICIAL INTELLIGENCE, 24. **Proceedings...** [S.l.: s.n.], 2012. p.11–18.

BENHAMOU, B. et al. Enhancing Clause Learning by Symmetry in SAT Solvers. In: IEEE INTERNATIONAL CONFERENCE ON TOOLS WITH ARTIFICIAL INTELLIGENCE, 2010. **Proceedings...** IEEE, 2010. p.329–335.

BRADEL, B.; ABDELRAHMAN, T. Automatic trace-based parallelization of java programs. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING. ICPP 2007. **Proceedings...** [S.l.: s.n.], 2007. n.Icpp.

BUYYA, R.; SILVA, L. Parallel Programming models and paradigms. In: **High Performance Cluster Computing**: architectures and systems. [S.l.: s.n.], 1999. p.4–27.

CHRABAKH, W.; WOLSKI, R. GridSAT: a chaff-based distributed sat solver for the grid. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2003. **Proceedings...** [S.l.: s.n.], 2003. p.1–13.

CHU, G.; STUCKEY, P.; HARWOOD, A. Pminisat: a parallelization of minisat 2.0. **SAT race**, [S.l.], 2008.

CORDES, D.; MARWEDEL, P.; MALLIK, A. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In: IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE

CODESIGN AND SYSTEM SYNTHESIS (CODES+ISSS), 2010., New York, New York, USA. **Proceedings...** ACM Press, 2010. p.267.

Cuda C Best Practices Guide. [S.l.: s.n.], 2013.

CUDA C Programming Guide. [S.l.: s.n.], 2013.

DELEAU, H.; CHRISTOPHE, J.; KRAJECKI, M. GPU4SAT: solving the SAT problem on GPU. Available at: http://para08.idi.ntnu.no/docs/submission_49.pdf. Access on: January 20th, 2014.

DIAZ, J.; CARO, C. Muñoz; NINO, A. A survey of parallel programming models and tools in the multi and many-core era. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.23, n.8, p.1369–1386, 2012.

EÉN, N.; BIERE, A. Effective preprocessing in SAT through variable and clause elimination. In: THEORY AND APPLICATIONS OF SATISFIABILITY TESTING. **Proceedings...** [S.l.: s.n.], 2005. p.61–75.

EÉN, N.; SÖRENSSON, N. An extensible SAT-solver. In: THEORY AND APPLICATIONS OF SATISFIABILITY TESTING. **Proceedings...** [S.l.: s.n.], 2004. p.502–518.

FEIER, M. C.; LEMNARU, C.; POTOLEA, R. Solving NP-Complete Problems on the CUDA Architecture Using Genetic Algorithms. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED COMPUTING, 2011. **Proceedings...** IEEE, 2011. p.278–281.

FUJII, H.; FUJIMOTO, N. GPU Acceleration of BCP Procedure for SAT Algorithms. Available at: http://elrond.informatik.tu-freiberg.de/papers/WorldComp2012/PDP6166.pdf http://world-comp.org/p2012/PDP6166.pdf. Access on: January 20th, 2014.

GIL, L.; FLORES, P.; SILVEIRA, L. PMSat: a parallel version of minisat. **Journal on Satisfiability, Boolean Modeling and Computation**, [S.l.], v.6, p.71–98, 2008.

GOMES, C. et al. Satisfiability solvers. In: HARMELEN, F. van; LIFSCHITZ, V.; PORTER, B. (Ed.). **Handbook of Knowledge Representation**. [S.l.]: Elsevier, 2008. p.89–134.

GOMES, C. P.; SELMAN, B.; CRATO, N. Heavy-tailed distributions in combinatorial search. In: **Principles and Practice of Constraint Programming**. [S.l.]: Springer, 1997. p.121–135.

HAMADI, Y.; JABBOUR, S.; SAIS, L. ManySAT: a parallel sat solver. **Journal on Satisfiability, Boolean Modeling and Computation (JSAT)**, [S.l.], v.6, p.245–262, 2009.

HAMADI, Y.; WINTERSTEIGER, C. Seven Challenges in Parallel SAT Solving. **AI Magazine**, [S.l.], 2013.

HOLLDOBLER, S.; MANTHEY, N. A short overview on modern parallel SAT-solvers. In: INTERNATIONAL CONFERENCE ON ADVANCED COMPUTER SCIENCE AND INFORMATION SYSTEMS. **Proceedings...** [S.l.: s.n.], 2011. p.978–979.

HUANG, J. The Effect of Restarts on the Efficiency of Clause Learning. **IJCAI**, [S.l.], p.2318–2323, 2007.

IRFAN, A.; MANTHEY, D. Modern Cooperative Parallel SAT Solving. Available at: http://tools.computational-logic.org/content/pcasso/pcasso-POS-2013.pdf. Access on: January 20th, 2014.

JANICIC, P. URSA: a system for uniform reduction to sat. **arXiv preprint arXiv:1012.1255**, [S.l.], 2010.

JURKOWIAK, B.; LI, C. M.; UTARD, G. A Parallelization Scheme Based on Work Stealing for a Class of SAT Solvers. **Journal of Automated Reasoning**, [S.l.], v.34, n.1, p.73–101, Jan. 2005.

KATEBI, H.; SAKALLAH, K. A.; MARQUES-SILVA, J. P. Empirical study of the anatomy of modern sat solvers. In: **Theory and Applications of Satisfiability Testing-SAT 2011**. [S.l.]: Springer, 2011. p.343–356.

KLEINBERG, J.; TARDOS, É. **Algorithm design**. [S.l.]: Pearson Education India, 2006.

KOTTLER, S.; KAUFMANN, M. Sartagnan-a parallel portfolio sat solver with lockless physical clause sharing. **Pragmatics of SAT**, [S.l.], 2011.

KROENING, D.; STRICHMAN, O. **Decision procedures: an algorithmic point of view**. [S.l.: s.n.], 2008.

LEWIS, M.; SCHUBERT, T.; BECKER, B. Multithreaded SAT Solving. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2007. **Proceedings...** IEEE, 2007. p.926–931.

MADIGAN, C.; MOSKEWICZ, M.; MALIK, S. Efficient conflict driven learning in a Boolean satisfiability solver. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN. ICCAD 2001. IEEE/ACM DIGEST OF TECHNICAL PAPERS (CAT. NO.01CH37281). **Proceedings...** IEEE, 2001. p.279–285.

MARQUES-SILVA, J.; LYNCE, I.; MALIK, S. Conflict-driven clause learning SAT solvers. **SAT Handbook**, [S.l.], 2009.

MARTINS, R.; MANQUINHO, V.; LYNCE, I. Improving Search Space Splitting for Parallel SAT Solving. In: IEEE INTERNATIONAL CONFERENCE ON TOOLS WITH ARTIFICIAL INTELLIGENCE, 2010. **Proceedings...** IEEE, 2010. p.336–343.

MCDONALD, A. Parallel WalkSAT with Clause Learning. Available at: http://www.ml.cmu.edu/research/dap-papers/dap_mcdonald.pdf. Access on: January 20th, 2014.

MEYER, Q.; SCHONFELD, F.; SCHONFELD, F. 3-SAT on CUDA: towards a massively parallel sat solver. In: HIGH PERFORMANCE COMPUTING AND SIMULATION (HPCS), 2010 INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2010.

MOSKEWICZ, M.; MADIGAN, C.; ZHAO, Y. Chaff: engineering an efficient sat solver. In: DESIGN AUTOMATION CONFERENCE, 38. **Proceedings...** [S.l.: s.n.], 2001.

PADBERG, F.; MIROLD, M. An Experimentation Platform for the Automatic Parallelization of R Programs. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, 2012. **Proceedings...** IEEE, 2012. p.203–212.

PIPATSRISAWAT, K.; DARWICHE, A. On the power of clause-learning SAT solvers with restarts. **Principles and Practice of Constraint Programming-CP**, [S.l.], 2009.

QUAMMEN, C. Introduction to programming shared-memory and distributed-memory parallel computers. **Crossroads**, [S.l.], v.12, n.1, p.2–2, Oct. 2005.

SANDERS, J.; KANDROT, E. **CUDA by example**: an introduction to general-purpose gpu programming. [S.l.]: Addison-Wesley Professional, 2010.

SCHUBERT, T. et al. MiraXT - A Multithreaded SAT Solver. **System description for the SAT competition**, [S.l.], p.3–4, 2007.

SHACHAM, O.; ZARPAS, E. Tuning the VSIDS decision heuristic for bounded model checking. In: INTERNATIONAL WORKSHOP ON MICROPROCESSOR TEST AND VERIFICATION - COMMON CHALLENGES AND SOLUTIONS, 4. **Proceedings...** IEEE, 2003. p.75–79.

SILVA, J.; SAKALLAH, K. GRASP-a new search algorithm for satisfiability. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1996. **Proceedings...** [S.l.: s.n.], 1997.

SILVA, L.; BUYYA, R. Parallel programming models and paradigms. **High Performance Cluster Computing: Architectures and Systems**, [S.l.], p.4–27, 1999.

SINGER, D. Parallel resolution of the Satisfiability Problem: a survey. **Parallel combinatorial optimization**, [S.l.], 2006.

SINZ, C.; BLOCHINGER, W.; KüCHLIN, W. PaSAT - Parallel SAT-checking with lemma exchange: implementation and applications. **Electronic Notes in Discrete Mathematics**, [S.l.], 2001.

TEIGE, T.; HERDE, C.; FR, M. A generalized two-watched-literal scheme in a mixed Boolean and non-linear arithmetic constraint solver. In: EPIA 2007. **Proceedings...** [S.l.: s.n.], 2007.

The International SAT Competitions Web Page. Available in: http://www.satcompetition.org. Access on: January 20th, 2014.

XU, L. et al. SATzilla: portfolio-based algorithm selection for sat. **Journal of Artificial Intelligence Research (JAIR)**, [S.l.], v.32, p.565–606, 2008.

YUEN, H.; BEBEL, J. **Tough SAT Project**. Available at: http://toughsat.appspot.com. Access on: January 20th, 2014.

ZHANG, H.; BONACINA, M. P.; HSIANG, J. PSATO: a distributed propositional prover and its application to quasigroup problems. **Journal of Symbolic Computation**, [S.l.], v.21, n.4-6, p.543–560, Apr. 1996.

Appendices

APPENDIX A – RESUMO ESTENDIDO

SAT Solvers Paralelos e Suas Aplicações em Paralelização Automática

SAT Solvers são software utilizados para resolver o problema SAT. Data uma fórmula Booleana em CNF, eles são capazes de determinar se esta fórmula tem um conjunto de atribuições que faz a fórmula verdadeira. SAT Solvers são utilizados em diversas áreas e já apresentaram muitas melhorias nos últimos anos. Apesar disso, o problem SAT é um problem NP-Completo e não existe nenhum algoritmo que o resolva em tempo polinomial no pior caso. Se tal algoritmo for algum dia encontrado, isso provará que $P = NP$.

Introduzimos esta ideia num contexto de paralelização automática. Desde a diminuição da tendência de melhora na frequência de processadores, uma nova tendência surgiu para permitir que softwares tirem proveito de harwares mais rápidos: a paralelização. Contudo, diferente de aumentar a frequência de processadores, utilizar parallelização requer um tipo diferente de programação, a programação paralela, que é geralmente mais difícil que a programação sequencial comum. Neste contexto, a paralelização automática apareceu, permitindo que o software tire proveito do paralelismo sem a necessidade de programação paralela.

Nós apresentamos neste documento duas propostas: SAT-PaDdlinG e RePaSAT. A primeira dessas propostas, SAT-PaDdlinG é um SAT Solver paralelo capaz de resolver o problem SAT em um ambiente de GPU. Esse SAT Solver é um SAT Solver DPLL, um tipo de SAT Solver que já ganhou diversas otimizações, inclusive, essa é a primeira proposta de um SAT Solver DPLL paralelo para GPU. Com essa proposta, nós esperamos fornecer paralelismo massivo para aplicações que utilizam SAT Solvers, como RePaSAT.

RePaSAT é a nossa proposta de ferramenta de paralelização automática. Essa ferramenta traduz partes de código em fórmulas Booleanas que podem ser paralelizadas automaticamente por SAT Solvers paralelos. Isso nos traz o ganho de não necessitar de programação paralela, qua geralmente é mais difícil que programação sequencial e nos permite tirar proveito de ambientes paralelos. Em particular, gostaríamos que esta ferramenta tirasse proveito do ambiente de GPU, utilizando SAT-PaDdlinG, porém qualquer SAT Solver pode ser utilizado, dado que aceitem fórmulas CNF.

Inicialmente são apresentados um conjunto de conceitos utilizados para entender as propostas apresentadas. Conceitos básicos sobre SAT Solvers são apresentados, o procedimento básico apresentado por eles, incluindo fazer uma decisão, determinar implicações dessa decisão através de BCP, detectar conflitos e fazer *backtracking* por causa do conflito. Esses procedimentos são apresentados junto com um conjunto de características do problema.

São apresentados um conjunto de otimizações apresentados por SAT Solvers, de uma forma geral, otimizações implementadas por SAT-PaDdlinG. As otimizações apresentadas são:

- **SAT Solvers CDCL** : Estes SAT Solvers são capazes de detectar conflitos e gerar cláusulas aprendidas a partir desses conflitos. Essas cláusulas aprendidas ajudam o SAT Solver a detectar implicações e conflitos mais cedo na busca, tornando a busca mais curta. Essa característica é tida como o principal motivo de SAT Solvers serem capazes de resolver fórmulas grandes.
- ***Backtracking* não-cronológico** : Este tipo de *backtracking* permite ao SAT Solver voltar mais de um nível de decisão, ao determinar que um conflito tenha ocorrido. Neste cenário, o SAT Solver não precisa percorrer parte do espaço de busca, tornando a busca menor.
- ***Two-watched literals*** : Esta otimização permite ao SAT Solver acelerar o procedimento BCP.
- **Reiniciações** : Reiniciações são utilizadas para tirar SAT Solvers de áreas do espaço de busca que não contenham a resposta.
- **Heurísticas de decisão** : Heurísticas de decisão ajudam o SAT Solver a fazer decisões mais inteligentes, chegando mais cedo à solução.

Uma vez apresentadas as otimizações sequenciais, estratégias de paralelização de SAT Solvers presentes na literatura são apresentadas. Em particular, abordagens cooperativas e competitivas são mostradas. As abordagens apresentadas são: divisão-e-conquista estática, *task farm*, roubo dinâmico de trabalho e portfólio. Essas abordagens são utilizadas para atacar um dos principais problemas na paralelização de SAT Solvers: a desuniformidade do espaço de busca, que causa dificuldades de distribuição de carga.

Uma vez apresentados conceitos de paralelização de SAT Solvers, conceitos de CUDA são apresentados. CUDA é o ambiente em que programas para GPU são desenvolvidos. Como SAT-PaDdlinG é feito em CUDA, estes são conceitos importantes.

Trabalhos relacionados são apresentados. SAT Solvers paralelos em CPU são apresentados com o objetivo de verificar a possibilidade de utilizá-los com RePaSAT. Como

buscamos paralelização massiva, esses SAT Solvers não fornecem a paralelização esperada, pois ou não apresentam paralelização massiva ou apresentam paralelização massiva em *clusters*, uma forma cara de paralelizar.

Apresentamos, então, SAT Solvers paralelos em GPU. Porém, muitos desses SAT Solvers não tem seus códigos-fonte disponíveis e os que têm apresentam características que impossibilitam seu uso para o nosso objetivo. Dessa forma, justificamos a criação de nosso próprio SAT Solver.

Apresentamos também outras ferramentas de paralelização automática. Cada uma tem sua vantagem, mas nenhuma faz o que RePaSAT faz: traduz o código para uma entidade completamente paralelizável, no caso, uma fórmula SAT.

Temos discriminado a implementação do SAT Solver SAT-PaDdling. Este SAT Solver implementa diversas otimizações, o tornando um SAT Solver capaz de resolver fórmulas grandes. Ele é um SAT Solvers CDCL, implementa *backtracking* não-cronológico, *two-watched literals* e a heurística de decisão VSIDS.

Algums experimentos são apresentados para determinar características deste SAT Solver. Os experimentos apresentados são os seguintes:

- ***Speedup*** : Esse experimento de *speedup* tem como objetivo determinar se SAT-PaDdlinG é capaz de paralelizar fórmulas com ganhos. De fato, ganhos são identificados e é mostrado como essa paralelização tem vantagens.
- **Tempo médio e tempo máximo de *thread*** : Esse experimento tem como objetivo testar se SAT-PaDdlinG contém problemas de distribuição de carga, comparando o tempo médio de *threads* com o tempo máximo. Problemas de distribuição de carga foram encontrados.
- ***Threads* vs blocos** : Uma comparação entre paralelizar com *threads* e blocos é feita. É determinado que a paralelização simples com *threads* dentro de um blocos não traz ganho e o número de blocos deve ser aumentado para paralelização. Isso mostra que não conseguimos tirar proveito da característica SIMT do GPU, mas mesmo assim conseguimos ter ganhos quando paralelizando com blocos.
- **Limite de blocos** : Neste experimento é mostrado como existe um limite para a paralelização em CUDA utilizando blocos, um limite de 16 blocos.
- ***Overhead* de paralelização** : Este experimento mostra o *overhead* de paralelizar código com SAT-PaDdlinG. É demonstrado que o *overhead* diminui com o tamanho da fórmula, o que faz esse *overhead* não impedir paralelização.

Por último, é apresentado a proposta de RePaSAT. RePaSAT é a proposta de paralelizador automático utilizando SAT. É mostrado como esta ferramenta faz para repre-

sentar estruturas de dados em uma fórmula CNF. As estruturas que esta ferramenta pode representar são:

- **Variáveis Booleanas** : Mesmo tipo de variáveis representadas em uma fórmula. A conversão é direta.
- **Variáveis inteiras** : Variáveis inteiras podem ser representadas como um vetor de variáveis Booleanas.
- **Variáveis de ponto fixo** : Variáveis de ponto fixo também podem ser representadas como um vetor de variáveis Booleanas, com diferentes operações.
- **Vetores** : São representados como um conjunto de variáveis de algum tipo.
- **Matrizes** : São representados como um conjunto de vetores.

São mostradas as conversões de código para fórmula SAT, de forma a ser paralelizados. Operações Booleanas e aritméticas (inteiras e de ponto fixo) podem ser traduzidas. Condicionais também e atribuições, mas não laços cujo número de iterações não é determinado em tempo de compilação.

Um conjunto de experimentos é feito para determinar a capacidade de paralelização de RePaSAT. Foi utilizado Ursa, *bash* e os SAT Solvers SAT-PaDdlinG e ManySAT para os experimentos. Os experimentos são:

- ***Speedup* com SAT-PaDdlinG** : Este experimento mostrou que SAT-PaDdlinG não apresenta melhoras em paralelos com fórmulas geradas por Ursa. Anteriormente, havia sido mostrado que SAT-PaDdlinG paraleliza bem fórmulas, então isso parece ser uma característica das fórmulas geradas por Ursa.
- ***Speedup* com ManySAT** : Este experimento tem como objetivo analisar se os resultados do experimento anterior eram só característica de SAT-PaDdlinG ou de outros SAT Solvers. Foi observado que as mesmas características acontecem como ManySAT, ele paraleliza entradas que não são geradas por Ursa, mas não paraleliza as geradas por esta ferramenta. Concluímos, assim, que essa é uma característica geral das fórmulas geradas por Ursa.
- ***Overhead* da técnica** : Comparamos uma execução sequencial de códigos traduzidos para fórmulas com códigos de C. Determinamos que há casos que RePaSAT é melhor e outros que C é melhor, mostrando que esta técnica provavelmente precisa ter um domínio específico.

Como conclusão final, não fomos capazes de paralelizar código com RePaSAT, mas mostramos muitas formas de transformar código em fórmulas SAT. Isso pode ter

várias vantagens no futuro. Além disso, esse trabalho pode ser continuado e outras alternativas analisadas. O principal problema já foi identificado e ele tem que ser resolvido antes da solução ser encontrada.

Nós fomos pioneiros em SAT Solvers DPLL para GPUs. Mostramos como rodar um SAT Solver em um ambiente SIMT funciona e como esse modelo deve ser evitado para SAT-PaDdlinG ter vantagens. Nós fornecemos um SAT Solver com várias otimizações e em GPU, que pode servir para diversos estudos e usos. Este SAT Solver pode ser muito melhorado, mas como está já fornece uma boa contribuição, já que paraleliza com sucesso várias fórmula em GPU.

APPENDIX B – OTHER EXPERIMENT WITH SAT PADDLING

We present here some another experiment carried out with SAT PaDdlinG. In Figure 1, we can see the time for the slowest thread of the following procedures: preprocessing, BCP with conflict analysis, backtracking and the decision time. In this case, the decision heuristic was a simple choise of the last free variable.

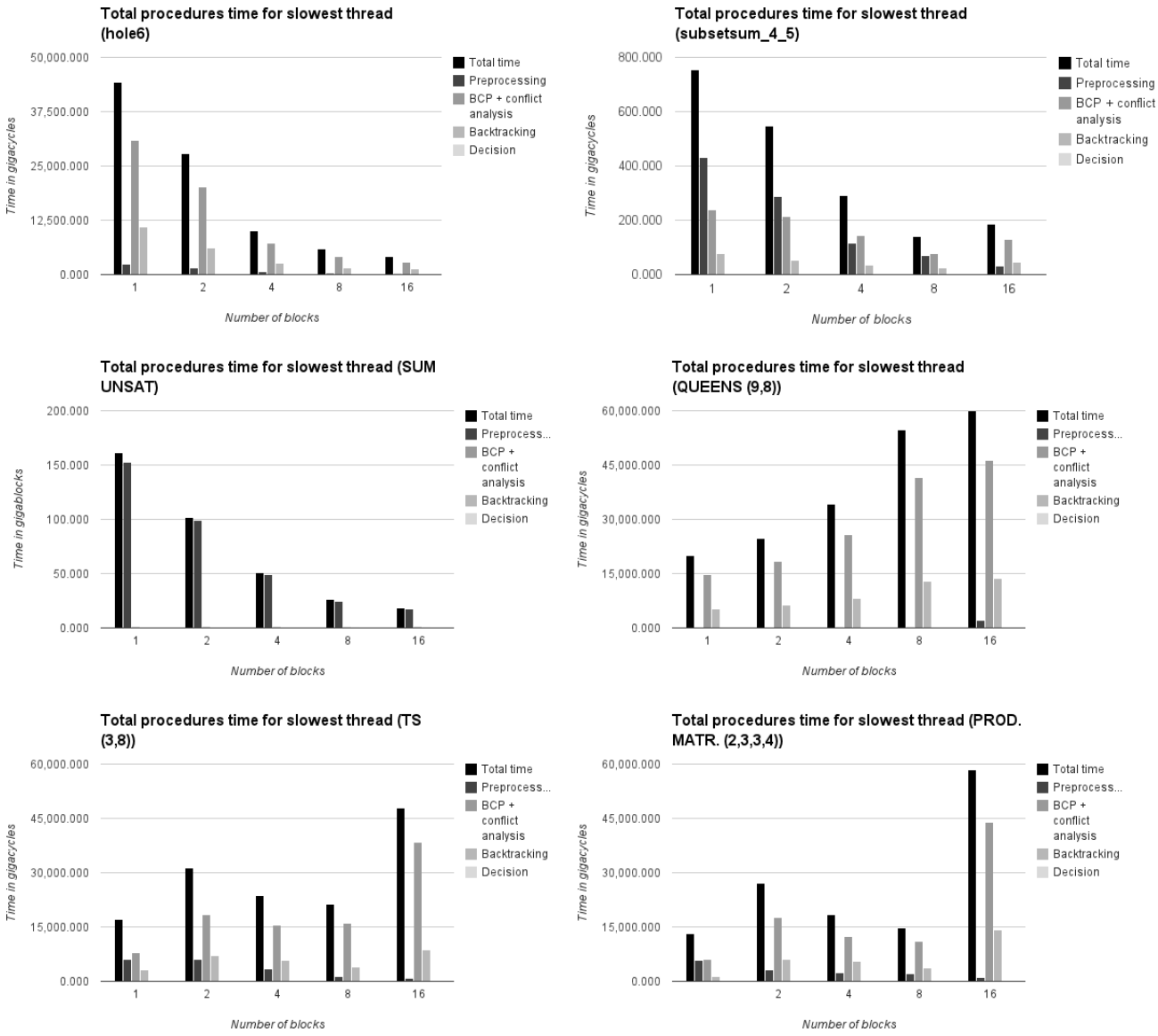
We can see many different characteristics. First of all, the decision time is very small when compared to the rest, which is expected, when considering that a very simple heuristic was used.

We can also see that in the smallest input, “SUM UNSAT”, in which the worst block takes a little more than 150,000 gigacycles to finish, that the preprocessing time consume most of the time for processing. In the second smallest, “subset_sum_4_5”, in which the total time almost reaches 800,000 gigacycles, the preprocessing is smaller, but still the largest procedure. The other inputs with larger total times took proportionally much less time in preprocessing when compared to their other procedures. This happens because the preprocessing stage time is dependent on the number of assumptions, which are processed in this step, and this number is exactly the number of variables per task. Since this number is fixed for all inputs in this experiment, its time does not change much for different inputs and its proportion to the rest of the procedures decreases as the formula’s size increases, because the other procedures increase.

We can also see that, in the cases where the preprocessing was not the largest procedure, the BCP + conflict analysis was. It is stated in (MOSKEWICZ; MADIGAN; ZHAO, 2001; SINGER, 2006) that in most cases, BCP consumes more than 90 % of the execution time. This characteristic seems to be true for the large formulas, we can see that this procedure is, in fact, very close to the total time in most cases. However, this value describes BCP and the conflict analysis together, which are executed together in SAT-PaDdlinG. Rerunning this experiment and gather separate values for these two procedures is a way to verify if in fact it is BCP that is consuming most of this time.

Backtracking also was not as high as the other procedures. It only happens when conflict happens, so we would not expect it to be as high as BCP, which happens after

Figure 1: Time spent by the most common procedures in DPLL by SAT PaDdlinG.



every decision. However, since it requires many structures to be updated (list of decisions and implications, implication graph, two-watched literals, etc.), it could bring some important overhead overall.