A New Approach to Parallel SAT Solvers

by

Max Nelson

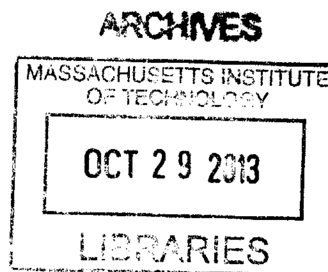
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013



© Massachusetts Institute of Technology 2013.
All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2013

Certified by ....
Armando Solar-Lezama
Associate Professor
Thesis Supervisor

Accepted by ....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

A New Approach to Parallel SAT Solvers

by

Max Nelson

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering

Abstract

We present a novel approach to solving SAT problems in parallel by partitioning the entire set of problem clauses into smaller pieces that can be solved by individual threads. We examine the complications that arise with this partitioning, including the idea of global variables, broadcasting global conflict clauses, and a protocol to ensure correctness. Along with this algorithm description, we provide the details of a C++ implementation, ParallelSAT, with a few specific optimizations. Finally, we demonstrate that this approach provides a significant speedup on a set of SAT problems related to program analysis.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor

Acknowledgments

Many thanks go out to the following people who helped in the making of this document, including: my family, especially Zak Nelson for answering many coding questions and Jody Nelson for help proofreading, Armando Solar-Lezama for help on pretty much everything, and the group of Justin Venezuela, Michael Mekonnen, Paul Woods, for study breaks at 158 Magazine Street.

Contents

1 Introduction **13**

1.1 Background 14

1.1.1 Serial SAT Solvers 14

1.1.2 Portfolio Parallel SAT Solver 15

2 Description of Algorithm **19**

2.1 Briefing 19

2.2 Partitioning and Global Variables 19

2.3 Global Variables 22

2.4 Search 23

2.5 Decision 24

2.6 Propagation 26

2.6.1 Handling global store conflicts 27

2.7 Analysis and Backtracking 32

2.8 Correctness Protocol 33

2.8.1 Ensuring global variable consistency 34

2.8.2 Search result consistency protocol 35

3 Implementation **45**

3.1 Briefing 45

3.2 Minisat 45
3.3 OpenMP 46
3.4 Partitioning algorithm 46
3.4.1 METIS Partitioner 48
3.4.2 Natural Partition 50
3.4.3 Creating the Set of Global Variables 50
3.5 Global Variable Representation 52
3.6 Clause and Message Communication 52
3.6.1 Clause and Message Format 55
3.7 Optimizations 56
3.7.1 Activity 56
3.7.2 Sleeping and Empty Read Buffers 57

4 Benchmarks 59

4.1 Briefing 59
4.2 Analysis 60
4.3 Scaling 66

5 Future Work and Conclusion 69

5.1 Future Work 69
5.2 Conclusion 70

List of Figures

1-1 A diagram of the major modules of a modern-day CDCL SAT solver 16
1-2 A sample portfolio architecture. 17

2-1 A sample partition with three threads 20
2-2 An example global conflict 30
2-3 Implication graph and cut 31
2-4 Resolution of a global conflict 31
2-5 Protocol for result and global variable consistency for each slave thread 41
2-6 The slave protocol in state machine form 42
2-7 Protocol for result and global variable consistency for the master thread 43
2-8 The master protocol in state machine form 44

3-1 A sample natural partition 51
3-2 The double buffer data structure 54
3-3 Messages 56

4-1 CDF of propagations for auction.sk:sat_SYN_1.cnf 63
4-2 CDF of propagations for auction.sk:sat_SYN_9.cnf 64

4-3 CDF of propagations for auction.sk:sat_SYN_17.cnf 64
4-4 CDF of propagations for auction.sk:sat_SYN_25.cnf 65
4-5 CDF of propagations for auction.sk:sat_SYN_33.cnf 65
4-6 Scaling to more threads 67
4-7 Scaling to more threads global count 68

List of Tables

2.1 A concrete partitioning example with 2 threads 21

4.1 Table of benchmarks for two threads 61

4.2 Propagations per benchmark 62

4.3 Comparison of two particular SAT instances 66

Chapter 1

Introduction

The Boolean satisfiability problem (SAT) is a classic computer science problem. Along with its many theoretical appearances, it appears in many application contexts, including electronic design automation (EDA) and model checking [MS08]. Unfortunately for these applications, SAT also has the well-known property of being an NP-complete decision problem. Therefore, a large amount of research has been put into reducing the exponential search space of SAT. Indeed, modern SAT solvers can handle large amounts of variables and clauses by exploiting the structure of the problem.

On a different note, computer architecture is currently going through "if not a revolution, certainly a very vigorous shaking-up" [HS08]. This shake-up is coming in the form of multi-core architectures. As we fit more and more transistors on a processor in accordance with Moore's law, we are still limited by their clock speed, which cannot be increased due to overheating. The current solution is to put more processors (cores) on a single chip which can communicate through shared hardware caches.

With these new parallel systems, researchers have sought to harness their power to solve SAT problems. We present a new approach to solving SAT problems in parallel using the idea of *partitioning* the problem into smaller pieces that can be solved by individual cores. We will first look at the influences of our SAT solvers, including sequntial SAT solvers and the current parallel approach. Next in chapter 2, we will describe the algorithm specifics, breaking it down into several modules. In chapter 3, we examine the actual C++ implementation of the above algorithm, showcased with a few optimizations that significantly boost performance. Chapter 4 will look at some benchmarks to see how our system fares versus modern state of the art solvers. Finally, chapter 5 will include some ideas for future work as well as some concluding remarks.

1.1 Background

1.1.1 Serial SAT Solvers

We will briefly describe the structure and workflow of a typical serial conflict-driven clause learning (CDCL) SAT solver, whose inner workings were initially proposed by the CHAFF solver [MMZ+01] and the enhanced implementation of the Davis-Putnam algorithm by Bayardo and Schrag [JS97]. Our motivation for this explanation is mainly that the implementation of our algorithm is built on top of an existing Minisat distribution, maintaining the same modules that compose a CDCL solver. For a more complete description, consider reading the original Minisat paper [ES03].

1. *Propagation* - Taken mainly from CHAFF, the propagation process in-

volves the setting of variables that are forced by the previously set variables. When a variable is forced to be set to two different values, we encounter a *conflict*, and enter the analysis module. In most SAT instances, the majority of the time in a CDCL solver is spent during propagation.

2. *Analysis* - Modern solvers build on the ideas presented by the GRASP algorithm [MSS96], in which a conflict is analyzed to produce a new learnt clause, and a state of assignments to revert the solver to, which is called *backtracking*. This new learnt clause is used to prune down the search space.

3. *Decision* - When there is no unit information to propagate, a variable, according to some set of heuristics, is set to some arbitrary value. We call this type of variable a *decision variable*. The number of decision variables in the current trail is referred to as the *decision level*.

The execution finishes when a top-level (decision level zero) conflict is found (unsatisfiable), or a satisfying assignment is found (satisfiable). A summary of these modules can be found in Figure 1-1.

1.1.2 Portfolio Parallel SAT Solver

The standard parallel SAT solver is built around the *portfolio* approach. The strategy behind this approach is to give each thread a copy of the entire SAT instance. Next, each thread searches on their own instance (usually with a different random seed or a different algorithm in an attempt to give each a different search space), and the system finishes when one thread has found either satisfiable or unsatisfiable. As an optimizaiton, as threads learn clauses, they can broadcast these clauses (typically if they are sufficiently small, as smaller

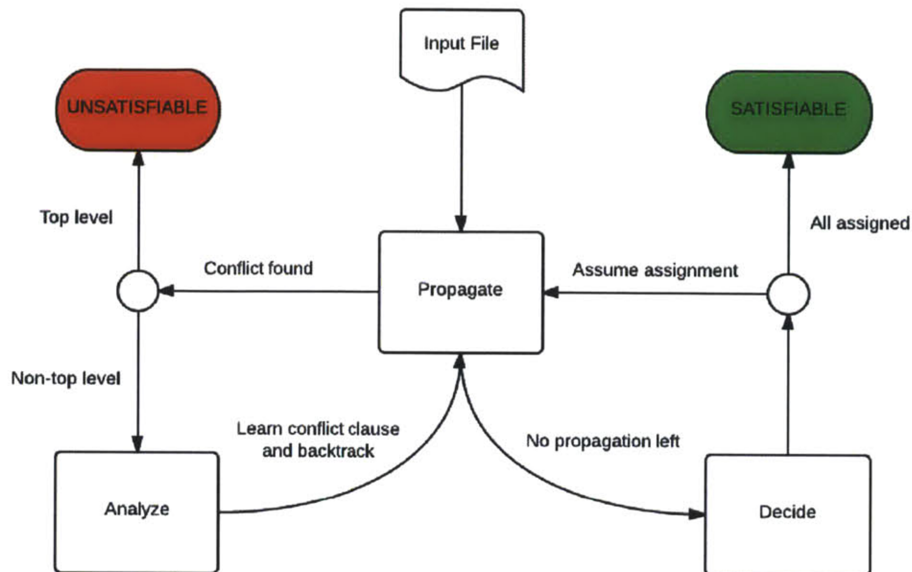


Figure 1-1: A diagram of the major modules of a modern-day CDCL SAT solver. Layout taken from http://www.mpi-inf.mpg.de/vtsa09/slides/leberre1.pdf.

clauses generally give more information on the instance) in order to help the other threads in their search process. A more in depth explanation can be found in the SATzilla paper [XHHLb].

The main problem with this approach is the space needed as we scale up the algorithm. As we add on more and more cores, we incur a $\mathcal{O}(n)$, where n is the number of threads, space cost as we are forced to copy the entire SAT instance into each solver. We will attempt to create a solver that is orthogonal to a portfolio solver such that it will be viable to run both approaches on the same cluster, achieving better results in terms of space and time than simply running one approach alone.

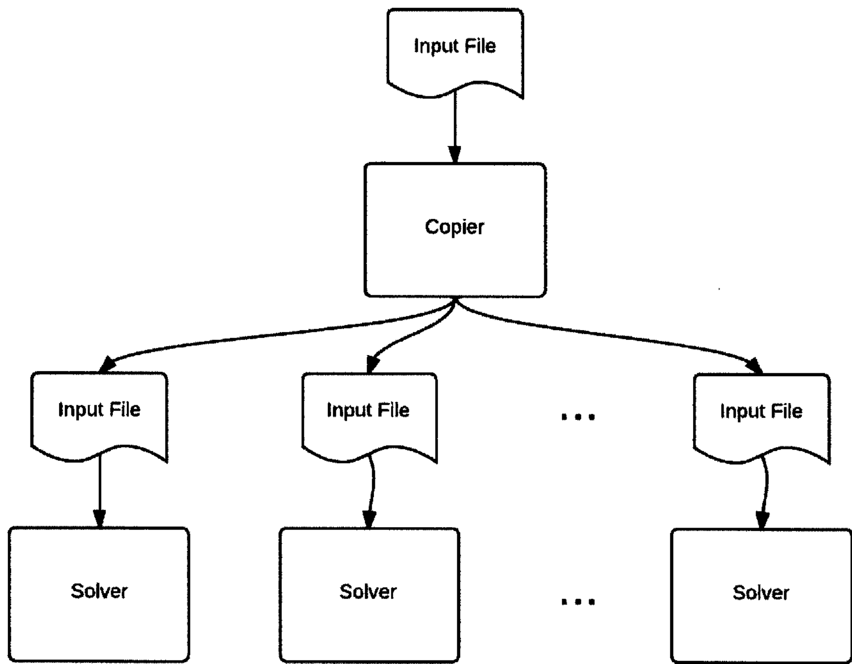


Figure 1-2: A sample portfolio architecture.

Chapter 2

Description of Algorithm

2.1 Briefing

Outlined below is the overall description of the algorithm. As mentioned above, our implementation is based on top of Minisat, so we will describe the algorithm in terms of the modules built into Minisat (and most modern SAT solvers).

2.2 Partitioning and Global Variables

For reference, consider the following SAT problem in conjunctive normal form (CNF): the conjunction (AND) of m clauses $\omega_1 \ldots \omega_n$, each of which is a disjunction (OR) of one or more literals, which is the occurence of one of n binary variables $x_1 \ldots x_n$ or its compliment.

The first step in the workflow is the *partitioner* module, which is not present in a sequential SAT solver. The partitioner's job is to create k partitions, where

k is equal to the number of threads desired by the user. Each k_i partition contains some disjoint subset of the m clauses which will be given to a particular thread as its own, smaller, SAT instance to solve.

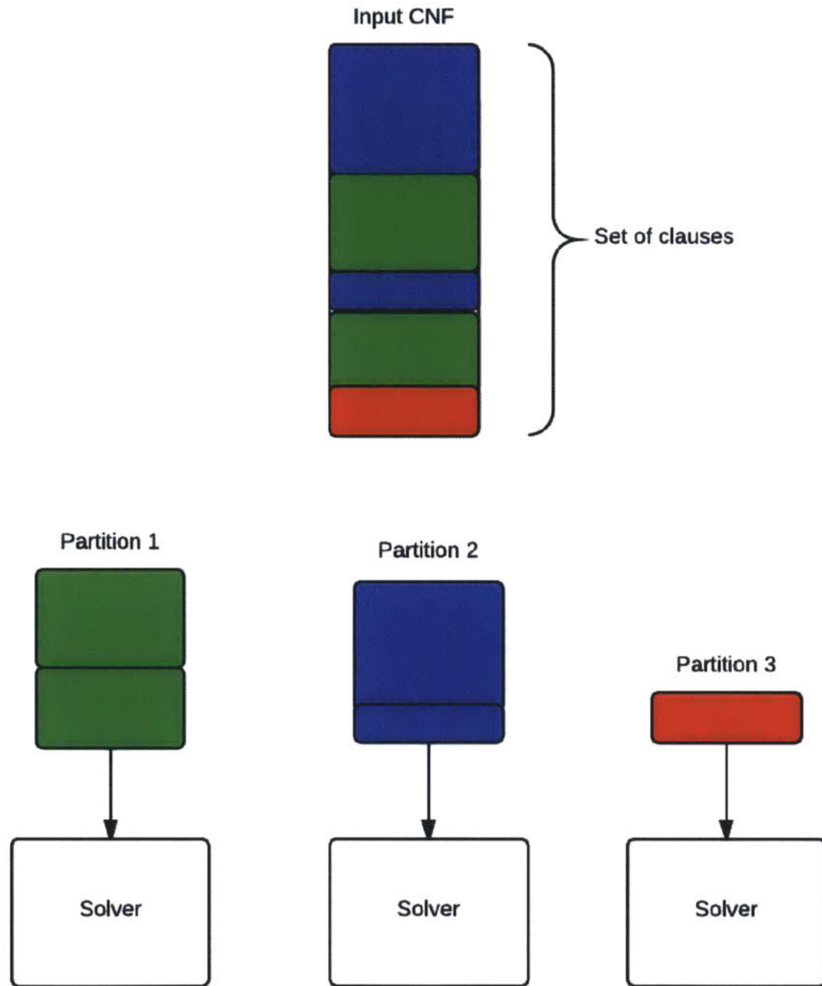


Figure 2-1: A sample partition with three threads.

By splitting up the problem into pieces, we hope to gain a speedup by having each thread work on a portion of the entire problem. If all partitions

are found to be satisfiable and are consistent in the assignment of all variables, then the entire instance is satisfiable, and if just one partition is found to be unsatisfiable, then the entire instance is unsatisfiable.

For example, consider the following small SAT problem, with only five clauses and four variables:

$$\omega_1 = x_1 \vee x_2$$
$$\omega_2 = \overline{x_1} \vee \overline{x_2}$$
$$\omega_3 = x_1 \vee \overline{x_2}$$
$$\omega_4 = \overline{x_1} \vee x_2$$
$$\omega_5 = x_3 \vee x_4$$
$$\omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 \wedge \omega_5$$

As the reader might notice, this SAT instance is unsatisfiable (due to clauses ω_1, ω_2, ω_3, ω_4, which in layman's terms requires that x_1 and x_2 be simultaneously the same and different values, which is, of course, impossible).

For simplicity, say we want to split this problem into two partitions (meaning we have two threads solving this SAT problem). A partitioning algorithm could split this problem as shown in Table 2.1, with thread 1 receiving $\omega_1 \wedge \omega_2 \wedge \omega_5$, and thread 2 receiving $\omega_3 \wedge \omega_4$.

Thread 1	Thread 2
$x_1 \vee \overline{x_2}$	$x_1 \vee x_2$
$\overline{x_1} \vee x_2$	$\overline{x_1} \vee \overline{x_2}$
$x_3 \vee x_4$	

Table 2.1: A concrete partitioning example with 2 threads

Now we encounter the fundamental dilemma with this new approach to parallel SAT solving. Thread 1 could find the satisfying assignment $x_1, x_2, x_3, x_4 =$ TRUE, and thread 2 could find the satisfying assignment $x_1 =$ TRUE, $x_2 =$ FALSE, thereby concluding that the entire instance is satisfiable, even though we clearly see that the instance is unsatisfiable. The problem, as the reader may notice, is that the threads have a different value for a variable that they share: x_2. The rest of the algorithm is designed around solving this problem.

2.3 Global Variables

For now, however, we will focus on the partitioner, and to do so we define two terms:

1. *Global variables*: variables that are in at least two partitions. In the above case, x_1 and x_2 are global variables.

2. *Local variables*: variables that are in exactly one partition. In the above case, x_3 and x_4 are local variables to thread 2.

In our algorithm, we have a shared *global store*, which is a mapping of global variables to their current value, either FALSE or TRUE. Each thread is able to read and write to this global store. Similarly, each thread has their own *local store*, where they store the value of the local variables. Much of the mechanisms in this algorithm are built around maintaining eventual consistency in each thread's view of the values of the global variables, ensuring that when we terminate, each thread agrees on the set of assignments.

In general, a good partitioning algorithm will attempt to accomplish two

goals: balance the work amongst the threads and minimize the number of global variables. The first of these goals is difficult: it is hard to determine which partitioning splits the work up evenly amongst the threads. The second of the these goals will be examined in more depth in later sections, but at a high level, minimizing global variables reduces the communication between threads, which can be a bottleneck in the algorithm. The number of global variables is a prime indicator of how fast this approach will run: zero global variables cuts the problem size in half for two partitions, while have every variable be global gives little benefit and slows down the solving through constant communication between threads.

We note that the rest of the algorithm does not depend on any notion on how the clauses are partitioned. Therefore, any algorithm to fit an application's needs can easily be interchanged into the system to optimize performance. The goal of this project was not to create a strong partitioning algorithm, so a library called METIS [KK95] was used. We also present a much simpler algorithm which creates a "natural" partition. Both of these will be explored in the Implementation section.

2.4 Search

The two policies that must hold during the search process are the following:

1. If a variable is declared a forced variable (set through propagation rather than decision), it must be implied only by the thread's own local trail and not be related to the global store.

2. Whenever a global variable is changed assignments in the global store

(setting to True from False, or set to False from True), a conflict clause consisting of only global variables must be broadcasted to all threads explaining why the current global store is incompatible with a satisfying assignment. This allows other threads to change their local trail accordingly.

The goals of these policies are twofold: any clauses learned by a thread are actually valid and are independent of the constantly changing global store, and at the end of the execution, each thread will have the same local value for each of the global variables.

2.5 Decision

The first module of the search algorithm that we will focus on is the *decision* module. As the reader may recall from above, when a solver no longer has any unit information to propagate, it chooses an unset variable and assigns it to an arbitrary value. This will either produce new unit information to propagate, cause a contradiction in the local trail, or find a satisfying assignment.

Modern day SAT solvers use many heuristics (the most popular being *activity*: how often a variable is seen in the search, usually during the learning process) to choose which variable is the next to be set in the decision process [MMZ$^+$01]. Our algorithm uses the same heuristics, but with one very important exception, all global variables are picked before any local variables. Note that this does not imply that all global variables are assigned values before any local variables are assigned values, as local variables can still be set through the propagation of global variables.

Why do we pose this added restriction of deciding global variables before local variables? The primary reason is associated with the learning module of the solver. When we find a contradiction in our set of assignments that some global variable g_i is forced to have two different values, then we are assured that the reason for this contradiction is entirely because of decisions of global variables. Thus, a learned clause produced by an analysis procedure can be composed of entirely global variables that we can in turn broadcast to other threads.

When we have chosen which variable, x_i, to set next and we want it to be set to an arbitrary value v. Our algorithm's response can be divided into four subcases:

1. *x_i is a local variable*: Setting x_i does not affect any other thread, therefore we can simply set x_i to v locally and continue as normal.
2. *x_i is a global variable*:
    (a) *x_i is not set in the global store*: We are the first to set this variable! Atomically set x_i to v in the global store and in the local trail and continue the search.
    (b) *x_i is set to v in the global store*: It just so happened that x_i is already set to the desired value, therefore we can set x_i to v locally and continue.
    (c) *x_i is set to $\overline{v}$ in the global store*: Because the decision process is arbitrary, we minimize the disturbance in the global store by taking

the value and setting x_i to $\overline{v}$ locally.

As shown in the above cases, the general theme of the decision module is that it doesn't particularly matter which value we decide to take when setting a new decision variable, so it is least intrusive to take the value already in the global store if possible.

2.6 Propagation

The *propagation* module of a solver is a function that propagates unit information as dictated by the set of clauses and current assignments.

Consider the situation when we discover through propagation that x_i must have the value v. Once again, we can characterize our algorithm with a case analyis:

1. *x_i is a local variable*: Similar to the decision module case, setting x_i has no affect on other threads, so we can simply set $x_i = v$ locally and continue propagation.

2. *x_i is a global variable*:

   (a) *x_i is not set in the global store*: Same as the decision module, this thread happened to be the first to want to set x_i, so it can atomically set $x_i = v$, without worrying about consistency among the threads.

   (b) *x_i is set to v in the global store*: Once again, because the value in the global store is the desired value, the thread can set $x_i = v$ locally and be consistent with the global store.

(c) *x_i is set to $\overline{v}$ in the global store*: Now we reach the first non-trivial case. We cannot simply take the value in the global store as we did in decision module and continue with propagation because then our local assignment would be invalid. We will examine this case in further detail below.

2.6.1 Handling global store conflicts

There are two scenarios when a local assignment is incompatible with the global store, during propagation as shown above, and during backtracking which will be discussed in the Analysis and Backtracking section. In either case, we must resolve the conflict. Specifically the thread can either backtrack to an earlier level and begin searching again with the global store's value or it can change the value of the variable in the global store and broadcast the "reason" why this variable had to be changed. Our algorithm adopts the second approach. We found the first approach could easily lead to infinite loops as each thread is waiting for the overall situation to change, which, of course, never does if no thread is changing the state of the global store.

Therefore, when a thread learns through propagation that the global store is inconsistent, it flips the variable's value in the global store (either from TRUE to FALSE or FALSE to TRUE) and *broadcasts* a clause explaining why this variable was forced to be switched. Now the question arises, what literals should the broadcasted clause contain? To answer this, we use the notion of an *implication graph* as first described by the GRASP SAT solver [MSS96]. As described in the above paper we can create an implication graph, I, by defining the vertices and edges as follows:

1. Each vertex in I corresponds to a variable assignment $x_i = v$.

2. The predecessors of x_i are the set of variables that forced that variables assignment. For example, if a thread had the clause $x_1 \vee x_2$, and $x_1 =$ FALSE, then $x_2 =$ TRUE is forced and there is an edge from x_1 to x_2. A decision variable, on the other hand, has no predecessors because there is no set of variables that forced its assignment; it was just an arbitrary choice.

3. An extra vertex corresponding to the conflict in the global store, which is the negation of some vertex already in I.

As "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver" [ZMM01] describes, any cut that separates the conflict variable and the decision variables of this graph will be a valid learned clause that is implied by the original SAT instance. However, we have the additional constraint that our clause must consist of strictly global variables. Therefore, we cannot simply take the minimum-cut (smaller clauses convey more information than larger clasues) like most modern SAT solvers do, as it may contain local variables.

The simplest solution, and the one that we selected, is to take the cut that contains the entire set of decision variables. Therefore our conflict clause will be of the form $\overline{d_1} \vee \overline{d_2} \ldots \vee \overline{d_n} \vee c_1$, where d_i is the set of decision variables and c_1 is the literal that is forced locally. Our choice of deciding global variables has vindicated itself now: if we find a conflict with the global store through propagation, then our set of decision variables is entirely global.

Claim: If we encounter a conflict in the global store while setting a global variable, g_i, through unit propagation, the set of decision variables is composed entirely of global variables.

Proof. Assume for the sake of contradiction that our set of decision variables is not entirely global. This, of course, implies that there is at least one local variable, L_i, in the set of decision variables. Since we are setting g_i through unit propagation, this implies that it was not set by the decision process. However, this violates our process of picking global variables first, as it would have been chosen before L_i. □

Let's look at an example for concreteness and clarity. Consider the following set and division of clauses with two threads (variables labeled with g_i and l_i are global and local variables, respectively).

Thread 1	Thread 2
$\overline{g_1} \vee \overline{g_2} \vee \overline{g_3} \vee l_1$	$g_1 \vee g_2$
$\overline{l_1} \vee \overline{g_4}$	$g_3 \vee g_4$
$\overline{g_3} \vee l_2$	

Consider the following trail for thread 1 (we adopt the convention that capital letters are decision variables while lower case letters are forced variables set through unit propagation): $G_1 = \text{TRUE}$, $G_2 = \text{TRUE}$, $G_3 = \text{TRUE}$, and $l_1 = \text{TRUE}$. As we can see by the clauses for thread 1, this forces the assignment $g_4 = \text{FALSE}$. However, suppose that thread 2 has already set $G_4 = \text{TRUE}$, as shown in Figure 2-2, in its search process.

Therefore, thread 1 will encounter an inconsistency in the global store. Thread 1 creates the implication graph shown in Figure 2-3. As proposed, we take

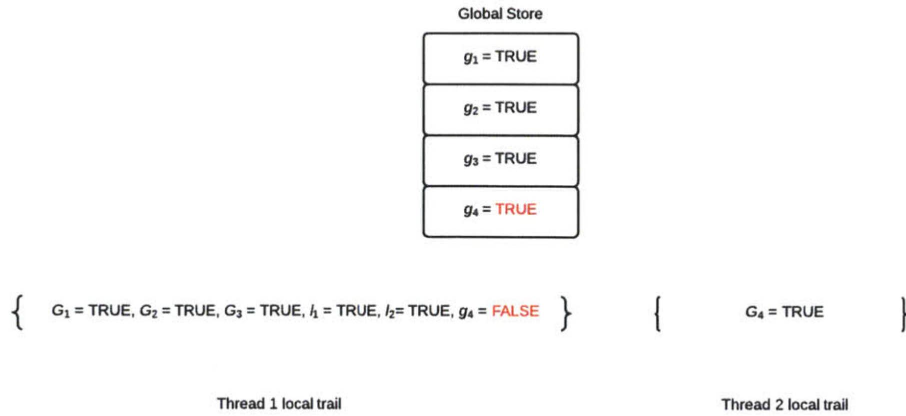


Figure 2-2: Thread 1 encounters that its local trail is incompatible with the global store.

the cut that includes the global conflict variable as well as the set of decision variables, producing the clause $\overline{g_4} \vee \overline{g_1} \vee \overline{g_2} \vee \overline{g_3}$. Intuitively, this clause can be seen as a clause representing the statement: our set of decision variable assignments AND g_4 being TRUE is NOT a compatible assignment. Using De Morgan's Law we can reduce that statement to the above clause.

Now, thread 1 flips the variable g_4 in the global store from TRUE to FALSE, adds the conflict clause to its own database and broadcasts the clause as the reason why this variable was forced to be flipped. Finally, thread 1 continues propagation as normal. A summary of this process can be found in Figure 2-4

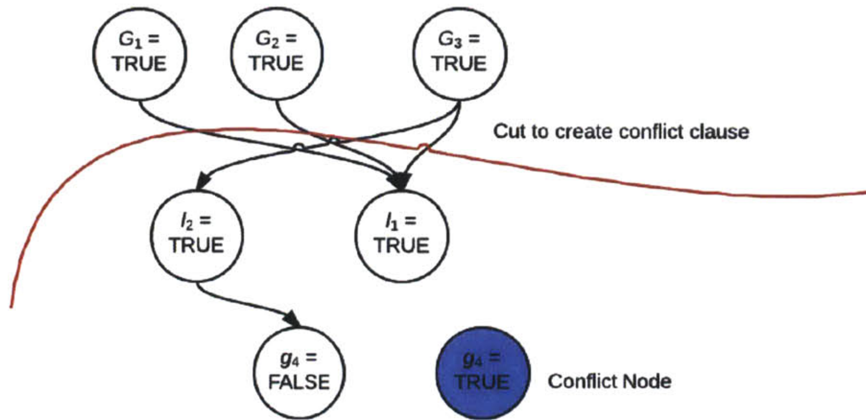


Figure 2-3: Thread 1's implication graph with the scenario described above. It takes the cut encompassing all of the decision variables, creating the conflict clause $\overline{g_4} \vee \overline{g_1} \vee \overline{g_2} \vee \overline{g_3}$.

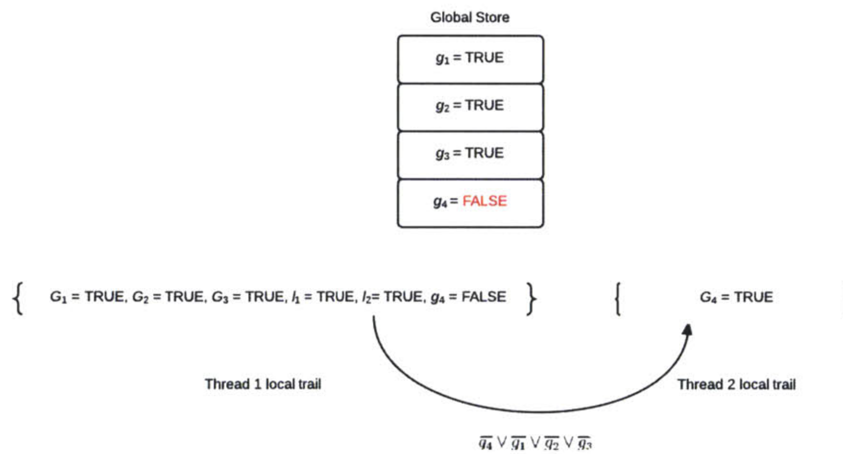


Figure 2-4: Thread 1 flips the value in the global store and broadcasts a clause explaining the switch.

2.7 Analysis and Backtracking

The analysis and backtracking module of Minisat and other SAT solvers is largely based off of ideas presented in the GRASP SAT solver [MSS96]. The workflow of this module is to first create a learned clause based on the conflict dictated by the set of decision variables, then backtracking to some decision variable and pushing it as forced due to the new learnt clause. Because conflict analysis does not effect the state of any variables in the system, we can use the same procedure without modification. However, the output of the analysis procedure, a conflict clause and a level to backtrack to, requires a little more attention.

Let ω_l be the learned clause created by the conflict analysis procedure, and k be the level that we should backtrack to according to this same procedure. Note that k is not necessarily equal to the current decision level minus one, as GRASP has shown, it is sometimes more efficient to jump back multiple decision levels at a time. Finally, let $d[k]$ be the decision variable at level k, and v be the value we will set to $d[k]$ after we backtrack.

As is customary by now, we split into cases (and subcases!):

1. $d[k]$ *is a local variable*: Backtracking a local variable has no bearing on the global store, so we can safely push $d[k]$ as forced with ω_l as the reason clause.

2. $d[k]$ *is a global variable*: We now want to set $d[k]$ to v in the local trail (as a forced variable), as well as in the global store. Note that this could involve changing the current value in the global store. As with all

cases when flipping a variable in the global store, we must broadcast a clause explaining why this variable was forced to be flipped. However, we encounter a subtle problem: the analysis procedure has no notion of local and global variables, and could produce a clause, ω_l, which contains a mixture of both. We must maintain the invariant that all broadcasted clauses consist entirely of global variables, so we divide into two subcases:

(a) ω_i *consists of entirely global variables*: It just so happened that our conflict analysis procedure produced an acceptable clause that we can broadcast. Atomically set $d[k] = v$ and broadcast ω_l.

(b) ω_i *is mixed with global and local variables*: While we can add this clause to our own local database, we cannot broadcast ω_l. Therefore, we need to create a new clause that explains why $d[k]$ is set to v in the global store. Fortunately, this is the same problem as creating a global conflict clause during propagation (Section 2.6.1), and we can use the exact same procedure to produce a new conflict clause ω_g. Atomically set $d[k] = v$ and broadcast ω_g.

Thus, we abide by our two invariants: whenever we backtrack and push a global variable as forced (thereby changing its value in the global store) it was entirely from our own set of local clauses and we broadcasted a clause explaining why it was forced to be changed.

2.8 Correctness Protocol

Throughout the search process, threads will concurrently be changing global variables as they attempt to find a satisfying assignment. At the end of the execution, all threads must have the same value locally for each global variable

compared to the global store and all threads must return the same result (either satisfiable or unsatisfiable). Therefore, we must develop a protocol for the threads to maintain eventual consistency, that is, at the end of the execution, these properties will hold true.

2.8.1 Ensuring global variable consistency

As described above in the propagation module, as a particular thread is propagating it may discover that it is forced to set a global variable to a different value than in the global store. To ensure progress and maintain consistency, we must broadcast a conflict clause describing why this variable has changed. This was also shown to be the case in the Analysis and Backtracking section.

When another thread receives such a clause, there must be a mechanism for the receiver to know which global variable was changed such that it is not forced to scan the entire global store to see if it is out of sync. While many mechanisms are feasible, we chose to make the first variable of the broadcasted conflict clause the global variable that was changed by the broadcaster. To make this more concrete, consider the following clause received by a thread:

$$\overline{g_1} \vee \overline{g_2} \vee g_3 \ldots \vee g_n$$

As mentioned above, the first variable, g_1, is designated to be the variable that flipped values in the global store by the broadcasting thread. At this point, we have three cases:

1. g_1 *is not in the receiver's trail*: In this case, the receiver can simply

add the broadcasted clause to its database and proceed with search as normal.

2. *g_1 is in the receiver's trail, but its value is the same as the value in the global store*: This can occur in executions where a variable is changed values many times before the broadcasted clause is actually assimilated by the receiver. Because the value is as the thread desires it, we can proceed with search as normal.

3. *g_1 is in the receiver's trail, and its value is different from the local store of the receiver*: The receiver must backtrack to the level in which this variable was set. Our implementation immediately sets $\overline{g_1}$ as a decision variable as an optimization, although this is not necessary.

Once again, we note that every time a global variable is flipped in the global store, a clause is broadcasted explaining the change. Therefore, as long as each thread receives every broadcasted clause, at the end of the execution every thread will have the same values for the global variables in their local trail.

2.8.2 Search result consistency protocol

Now we proceed to the problem of having each thread return the same value at termination: either satisfiable or unsatisfiable. At first glance this seems to be a rather simple protocol. However, developing a protocol in which each thread has equal responsibilities while still maintainining that each thread has the same eventual view of the global variables proved to be unwieldy and confusing. Thus, we settled on a master/slave protocol where one thread is designated the master and the rest are slaves. Before we delve into the respon-

sibilities of the parties, let us first introduce the different types of messages that are used in the protocol:

1. **FOUND_SATISFYING_ASSIGNMENT** : A message sent from the slave to the master indicating that they have found a satisfying assignment.

2. **FOUND_UNSATISFIABLE** : A message sent from the slave to the master indicating that they have found their instance to be unsatisfiable.

3. **STILL_FINISHED** : A message sent from the master to a slave asking whether the slave currently has found a satisfying assignment. This message has an a monotonically increasing sequence number associated with it.

4. **FINISHED_ACK** : A message sent from the slave to the master indicating that they still have a satisfying assignment. In other words, the slave has not received a broadcasted clause that forced it to backtrack. This message also has a sequence number attached to it that is equal to the sequence number of the **STILL_FINISHED** message that triggered it (explained in detail below).

5. **NOT_FINISHED_ACK** : A message sent from the slave to the master indicating that they no longer have a satisfying assignment. This can occur when a broadcasted clause forced them to backtrack.

6. **UNSATISFIABLE** : A message sent from the master to a slave indicating that the entire instance is unsatisfiable.

7. **UNSATISFIABLE_ACK** : A message from a slave to the master indicating that they have received the **UNSATISFIABLE** message.

8. **RETURN_SAT** : A message sent from master to a slave indicating that it is safe to return satisfiable.

9. **RETURN_UNSAT** : A message sent from master to a slave indicating that is is safe to return unsatisfiable.

The responsibilities of each party is as follows:

Slave: When a slave finishes its search process, it can either find satisfiable or unsatisfiable. We begin with the unsatisfiable case as it is more straightforward simply because if one thread finds unsatisfiable, then we know the entire instance is unsatisfiable, which is not true of a satisfiable result.

If a slave finds its partition to be unsatisfiable, the protocol is as follows:

1. Send a **FOUND_UNSATISFIABLE** message to the master.

2. Wait for an **UNSATISFIABLE** message from the master.

3. Send an **UNSATISFIABLE_ACK** message to the master.

4. Wait for a **RETURN_UNSAT** message from the master and return unsatisfiable.

Similarly, if any slave ever hears an **UNSATISFIABLE** message from the master, it responds with an **UNSATISFIABLE_ACK** and returns unsatisfiable when it receives a **RETURN_UNSAT** message.

Now we move on to the satisfiable case. As mentioned above, if one thread discovers a satisfying assignment for their partition, it does not necessarily mean the entire instance is satisfiable. Therefore, the system must ensure two

properties before it finally returns satisfiable: every thread has found a satisfying assignment and every thread has the same snapshot of values for the global variables in their local trail. With this in mind, the protocol for when a slave finds a satisfying assignment is as follows:

1. Send a **FOUND_SATISFYING_ASSIGNMENT** to the master.
2. If the slave begins searching again due to a broadcasted clause, send **NOT_FINISHED_ACK** to the master and begin searching again.
3. Wait for a **STILL_FINISHED** message (with some sequence number n) from the master.
4. If the slave is not currently searching, send back a **FINISHED_ACK** with sequence number n.
5. Wait for a **RETURN_SAT** message from the master and return satisfiable.

Note that it may be possible that a slave can send a **FINISHED_ACK** with a sequnce number n and not receive a **RETURN_SAT** message. This occurs when the master thread believes all threads are finished, but in reality some are still searching. Only in the case where each slave has responded to the same sequence number will the master send back the **RETURN_SAT** message (explained in more detail below).

The summary of the entire protocol for each slave thread can found in the pseudocode of Figure 2-5 and for the more pictorially inclined, examine Figure 2-6.

Master: The single master thread has more responsibilities than the slave threads. Its duties include tracking the state of the slave threads along with controlling when slave threads are allowed to return a value from their search. Once again, we will examine two cases, unsatisfiable and satisfiable results.

In the unsatisfiable case, we can break this down further into two subcases: A slave finds unsatisfiable (i.e, the master thread receives a **FOUND_UNSATISFIABLE** message) or the master thread itself finds unsatisfiable. In either case, the master thread uses the following protocol:

1. Send an **UNSATISFIABLE** message to each slave thread.

2. Wait for an **UNSATISFIABLE_ACK** message from each slave thread.

3. Send a **RETURN_UNSAT** message to each slave thread and return unsatisfiable.

The satisfiable case is a bit more involved. For this case, the master thread needs to store two values: an integer sequence number, *seq_num*, that is monotonically increasing and an array of boolean values, *finished_threads*, of length equal to the number of threads. As the name might indicate, if *finished_threads*$[i]$ is set, then the master thread believes thread i has found a satisfying assignment (note that this implies that the threads are numbered from 0 to $n-1$). Using these data structures, the protocol is as follows:

1. Set *finished_threads*[*my_thread_id*] to TRUE.

2. If you hear a **FOUND_SATISFYING_ASSIGNMENT** message from thread i, set *finished_threads*$[i]$ to TRUE.

3. If the slave begins searching again due to a broadcasted clause, set *finished_threads*[*my_thread_id*] to FALSE and begin searching again.

4. When all bits are set in *finished_threads*, send a **STILL_FINISHED** message to each slave with sequence number equal to *seq_num*, then increment *seq_num*.

5. If you hear a **FINISHED_ACK** from each slave with the correct sequence number, send **RETURN_SAT** to each slave and return satisfiable.

6. If you hear a **NOT_FINISHED_ACK** from thread i, set *finished_threads*[i] to FALSE and go to step 2.

An explanation of the above protocol is probably in order. The general strategy is that the master thread waits until it believes all threads, including itself, have found a satisfying assignment. When it has heard from all threads, it sends out a **STILL_FINISHED** message asking if the slaves are truly finished. If all threads agree to a particular sequence number, then we know that all threads have the same set of values for the global variables, and we are safe to finish.

The summary of entire protocol for the master thread can be found in the pseudocode of Figure 2-7.

```
1: function SLAVESEARCH
2:     loop
3:         if find unsatisfiable then                          ▷ I found unsat!
4:             Send FOUND_UNSATISFIABLE to the master thread
5:             while waiting for UNSATISFIABLE from the master do
6:                 sleep
7:             end while
8:             Send UNSATISFIABLE_ACK to the master thread
9:             while waiting for RETURN_UNSAT from the master do
10:                sleep
11:            end while
12:            return unsatisfiable
13:        end if
14:        if hear UNSATISFIABLE from the master then   ▷ Someone else found
   unsat!
15:            Send UNSATISFIABLE_ACK to the master thread
16:            while waiting for RETURN_UNSAT from the master do
17:                sleep
18:            end while
19:            return unsatisfiable
20:        end if
21:        if find satisfiable then
22:            Send FOUND_SATISFYING_ASSIGNMENT to the master thread
23:            while waiting for STILL_FINISHED message from the master
   thread do
24:                Read messages and begin searching again if read clause, send
   NOT_FINISHED_ACK to master
25:            end while
26:            Send FINISHED_ACK to the master thread
27:            while waiting for RETURN_SAT do
28:                sleep
29:            end while
30:            return satisfiable
31:        end if
32:    end loop
33: end function
```

Figure 2-5: Protocol for result and global variable consistency for each slave thread.

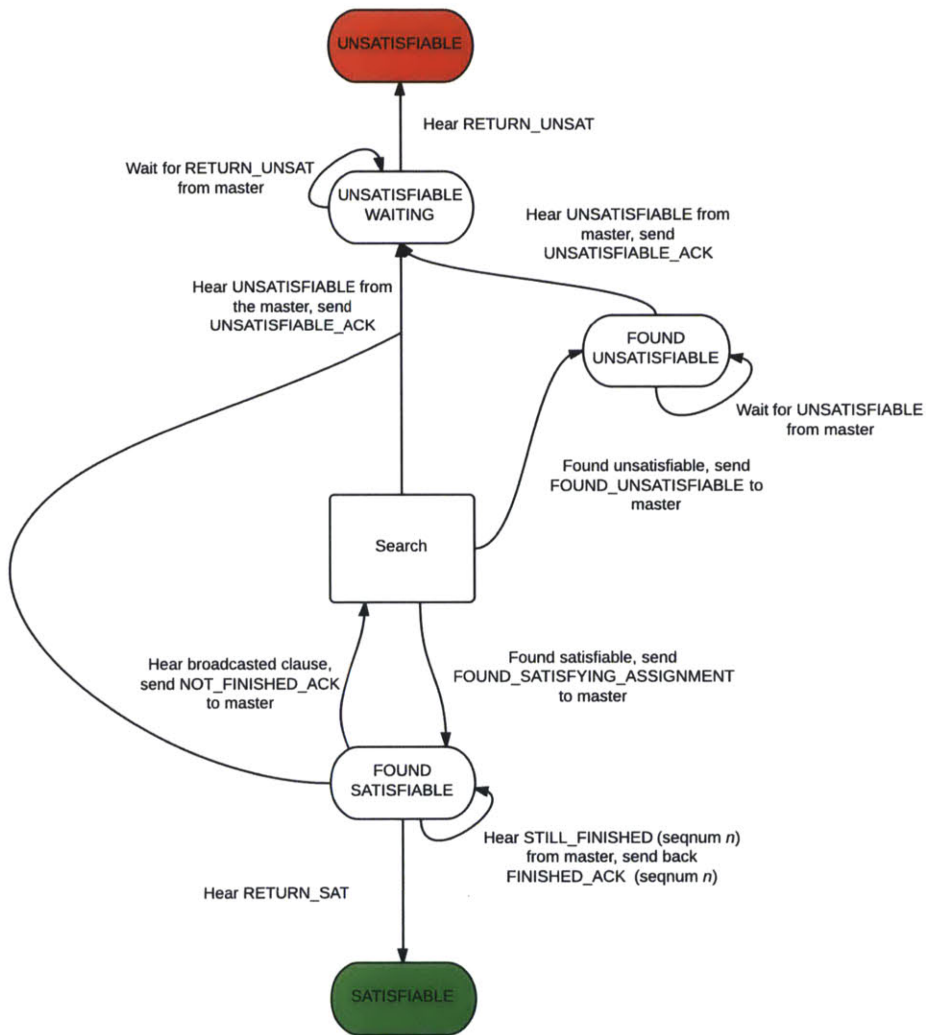


Figure 2-6: The slave protocol in state machine form.

```
1: function MASTERSEARCH
2:     loop
3:         if find unsatisfiable OR hear FOUND_UNSATISFIABLE then
4:             send UNSATISFIABLE to each slave thread
5:             while wait for UNSATISFIABLE_ACK from each thread do
6:                 sleep
7:             end while
8:             send RETURN_UNSAT to each slave thread
9:             return UNSATISFIABLE
10:        end if
11:        if find satisfiable assignment then
12:            finished_threads[my_thread_id] ← TRUE
13:        end if
14:        if FOUND_SATISFYING_ASSIGNMENT from thread i then
15:            finished_threads[i] ← TRUE
16:        end if
17:        if NOT_FINISHED_ACK from thread i then
18:            finished_threads[i] ← FALSE
19:        end if
20:        if all booleans are set in finished_threads then
21:            send STILL_FINISHED message to each slave
22:            seq_num ← seq_num + 1
23:            num_finished = 1                              ▷ I am finished!
24:            loop                                ▷ Now play the waiting game!
25:                if num_finished == NUM_THREADS then
26:                    send RETURN_SAT to each slave
27:                    return SATISFIABLE
28:                end if
29:                if FINISHED_ACK from slave with sequence number ==
    seq_num - 1 then
30:                    num_finished ← num_finished + 1
31:                end if
32:                if NOT_FINISHED_ACK from thread i then
33:                    finished_threads[i] ← textscFalse
34:                    break out of loop
35:                end if
36:            end loop
37:        end if
38:    end loop
39: end function
```

Figure 2-7: Protocol for result and global variable consistency for the master thread.

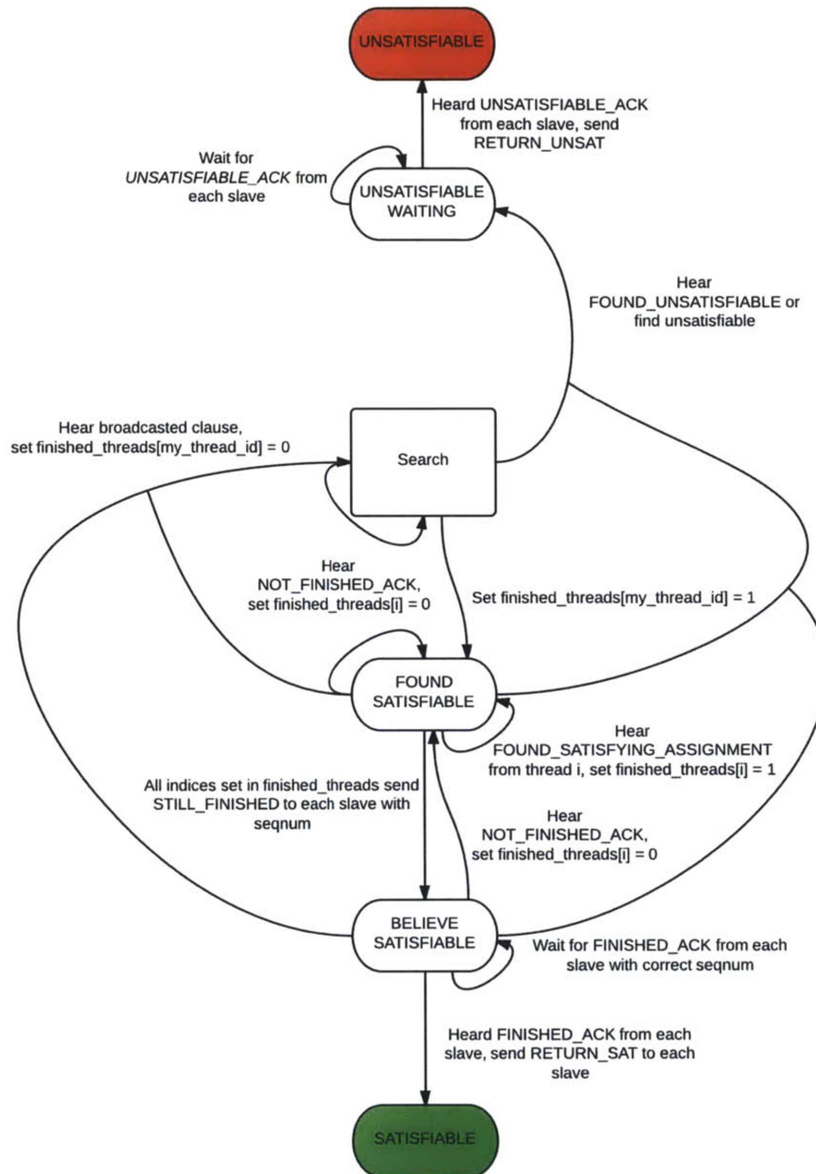


Figure 2-8: The master protocol in state machine form.

Chapter 3

Implementation

3.1 Briefing

With a better understand of the high level view of the algorithm, now we examine the actual implementation. The code was written in C++/Python and utilizes Minisat and OpenMP as the main foundations, both of which are described below. It can be cloned at https://bitbucket.org/maxnelso/parallelsat.

3.2 Minisat

The algorithm is built on top of the most popular, robust, and open-source SAT solver, Minisat, first created by Niklas Eén and Niklas Sörensson in 2003 [ES03]. Despite its age, it is still world class in terms of performance, and new iterations come out regularly with new optimizations. The International SAT competition holds Minisat in high regard: there is a special category called "Best Minisat Hack", which is for solvers that are built on top of Minisat.

Perhaps more important than its speed, however, is the design of Minisat. It was created with the idea that others would extend it with new optimizations. Therefore, it has clearly defined methods and data structures for all the standard pieces of a SAT solver: clauses, variables, propagation, conflict analysis, etc. Indeed, most of the implementation, excluding the message passing system and global variable system, were simply modifications to existing functions within Minisat.

3.3 OpenMP

The other main foundation we build upon is the C++ multithreading library called OpenMP [DM98]. OpenMP provides a simple API for all things multithreading: creation and deletion of threads, mutual exclusion mechanisms like locking and wrappers around critical sections, along with other utility funcutions, such as memory fences for flushing variable states and "barriers", which wait for all threads to reach a point in the code before proceeding. Despite being more "high-level" than a more fine-grained alternative such as POSIX Threads (Pthreads), OpenMP is known for being extremely performance conscious. The OpenMP infrastructure is so widespread that it was recently was proposed to become integrated with the C++ standard for language level parallelization.

3.4 Partitioning algorithm

The partitioner implements a Partitioner interface by completing one method PARTITION(STRING *path*, INT *num_threads*, SOLVER *solvers[]*). This method must prepare the *solvers*[] array for search by creating the clause database for

each solver.

To begin talking about our implementation of the partitioner (once again, this can easily be swapped out for some application specific partitioner), we must discuss the input to ParallelSAT, the DIMACS file, which describes a SAT instance. Taken from (http://people.sc.fsu.edu/ jburkardt/data/cnf/cnf.html), the grammar for a DIMACS file (with the slight ParallelSAT modifications) is as follows:

1. The file may begin with comment lines. The first character of each comment line must be a lower case letter "c". Comment lines typically occur in one section at the beginning of the file, but are allowed to appear throughout the file.

2. The comment lines are followed by the "problem" line. This begins with a lower case "p" followed by a space, followed by the problem type, which for CNF files is "cnf", followed by the number of variables, followed by the number of clauses.

3. The remainder of the file contains lines defining the clauses, one by one. A clause is defined by listing the index of each positive literal, and the negative index of each negative literal. Indices are 1-based, and for obvious reasons the index 0 is not allowed.

4. The definition of a clause may extend beyond a single line of text.

5. The definition of a clause is terminated by a final value of "0".

6. The file terminates after the last clause is defined.

Furthermore, we add the restriction that the variables are named from $1 \ldots V$, where V is the number of variables.

The partitioner converts this DIMACS file into n partitions, where n is the number of threads as specified by the user. Next, it adds clause ω_k $solvers[i]$ database if ω_k is in partition i.

However, this is not sufficient, as a thread also needs to know *which* variables are global. We adopt the convention that the first g variables are designated as the global variables. Thus, if there are 45 global variables with two threads, the variable 16 in the Minisat representation would be a global variable, while the variable 51 would be a local variable to that particular instance. Note that Minisat labels its variables from 0 to $k-1$, so 44 would be the last global variable in the above example. This also has the added benefit that in the Minisat internal representation of variables, we can easily check if a variable v is global through the check $v < g$.

Now that we know what the general workflow of one of these partitioners, let's examine the two partitioners we used in our implementation: a partitioner using the library METIS and a more simpler "natural" partitioner algorithm.

3.4.1 METIS Partitioner

METIS is a library that takes in a graph, G, as input and outputs a file that assigns each node in G to a partition. METIS was designed for parallel computing, and its goal is to balance the partitions as much as possible as well as minimizing the "connections" between the partitions, which requires interpro-

cess communication. As one might guess, the "connections" in our case are the global variables.

We will not go into the specifics of the input/output format of METIS, as that can be found in excellent detail in their documentation [Kar]. However, the question of what G is composed of must be addressed. The original algorithm was to create G as follows:

1. $\forall$ clause ω_i in the SAT instance S, create a node u_i.
2. $\forall$ pair of clauses ω_i and ω_j in S, create an edge between u_i and u_j if and only if there exists a variable, v_k, such that $v_k \in \omega_i \wedge \omega_j$.

Clearly by minimizing the connections in partitions of G, we minimize the global variables. While this produced better partitions than our current algorithm, for larger problems, G became too large (as there are $\mathcal{O}(n^2)$ edges, where n is the number of clauses), and METIS was incapable of finishing in a reasonable amount of time. However, if the problem is small enough or has the correct properties, the above algorithm may be a better choice than the revised algorithm that is presented below.

The algorithm we actually use to produce G is as follows:

1. $\forall$ variable v_i in the SAT instance S, create a node t_i. Call this set of nodes T.
2. $\forall$ clause ω_i in S, create a node u_i. Call this set of nodes U.
3. $\forall$ pair of nodes t_i, u_i in T, U (respectively), create an edge between t_i and u_i if and only if variable $v_i \in$ clause ω_i.

Note that this bipartite graph approach scales in $\mathcal{O}(n * m)$, where n is the number of clauses and m is the number of variables in the SAT instance. This has shown to speed up the partitioning process, and METIS will finish in a reasonable amount of time even for very large problems.

Thus, we pass this graph into METIS which outputs a file that assigns each node to a partition. Note that the partitioning of the t_i nodes are of no use to us, as we only care about the partitioning of the clauses.

3.4.2 Natural Partition

For some SAT problems, there is plenty of locality within the input DIMACS file. Therefore, one effective algorithm is to simply divide the input file into n parts, where n is equal to the number of threads. We call this a *natural partition*. While this doesn't produce as good of partitions as the METIS approach, actually producing the partition is magnitudes faster, leading to an overall faster system. The benchmarks we examined use this partitioning algorithm as an example of a sample application where custom partitioners are useful. An example natural partition with four threads can be found in Figure 3-1.

3.4.3 Creating the Set of Global Variables

In both algorithms, the next step is to figure out the set of global variables. The procedure is simply the brute force version: iterate through the entirety of the variables, and if the variable appears in more than one partition, then it is a global variable. Fortunately, we can build up the following hash map as we are parsing the inital DIMACS file:

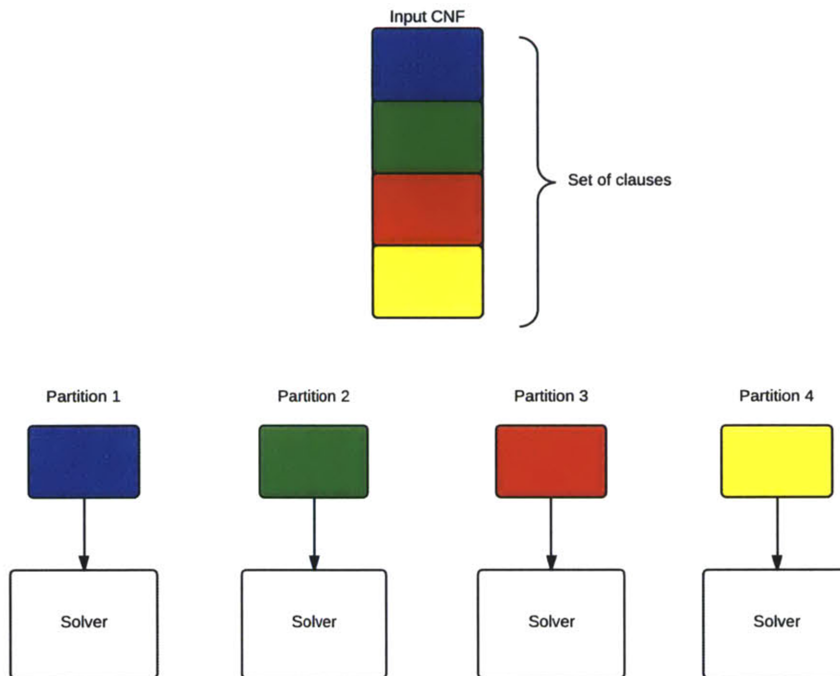


Figure 3-1: A sample natural partition with four threads.

$$\textit{var_to_clauses}\text{: } v_i \rightarrow \text{set of clauses } \omega_i \text{ s.t } v_i \in \omega_i$$

Which greatly simplifies the global variable detecting process.

We aren't quite finished, though, as we must complete the final mapping of global variables to the first g variables of the solver instances. Note that this may leave gaps in certain partitions. For example, partition 1 and partition 2 could share some variable g_i, but partition 3 does not have a clause that contains g_i. Therefore, we must give special care in each solver instance to still allocate space for this missing variable, otherwise a local variable could be mistaken for a global variable.

3.5 Global Variable Representation

Now we proceed into the search implementation, beginning with how the global variables are represented in the system. Minisat represents variables with an integer from $0 \dots n-1$. We adopt the same strategy, with the additional information that the first g variables are global variables. We represent the state of these variables with a simple array of length g, named *global_vars*, that contains the current state of global variables (either TRUE, FALSE, or UNSET). Because we allocate the first g variables to be global variables, we can set g_i (where g_i is an integer as represented in Minisat) by setting the value in $global_vars[g_i]$. To avoid reader-writer problems, we also need a lock on each global variable. Thus, we have another array named *global_vars_locks*, that is a one-to-one mapping to the *global_vars* array. By having an array of locks, we reduce contention across the threads for the price of a small memory overhead. A thread must acquire the lock for a particular global variable before it changes its value.

3.6 Clause and Message Communication

The clause and message communication module is one of the most important pieces of the system. First and foremost are the many learned clauses that threads pass around to each other, but the system also must handle the correctness protocol (Section 2.8). Therefore, an efficient implementation in this area is paramount. The interface for the system should be simple: we have the following operations (optional arguments are enclosed in braces):

1. READMESSAGES()

2. SENDCLAUSE(CLAUSE* *clause*)

3. SENDMESSAGE(MESSAGETYPE *msg_type*, [INT *seq_num*], [INT *thread_recipient*]).

The main data structure involved in the communication system is the *double buffer*, which as the name suggests, is two buffers. The general idea behind the double buffer data structure is to reduce contention amongst threads: writers are working on one buffer, while the reader is working on a completely different buffer. Each thread contains exactly one of these double buffers which acts as its inbox for incoming messages. Therefore, every thread has a pointer to every other thread's double buffer. Let us examine the double buffer of thread t_i, with b_1 and b_2 being the two buffers that compose the double buffer.

Initially, both b_1 and b_2 are empty. We begin in state **WRITE_B1** , which means that other threads are writing clauses and messages to b_1. When b_1 becomes full with clauses and messages (currently the size is 1024 variables), the writer waits for b_2 (the read buffer) to be empty, indicating the owner of the double buffer has read all of the incoming messages. When it is empty, the writer performs a *swap operation*: it sets the state of the buffer to **WRITE _B2** and writes its message to b_2, the new write buffer. Now the owner of the buffer reads messages from b_1.

Note that there are some concurrency issues here. For example, two threads could be attempting to write to the same write buffer at the same point and the result is garbled. Therefore, we need to protect these critical regions with locks. Specifically, whenever a thread attempts to add a clause to the write buffer (which involves checking how full the buffer is and the actual writing to

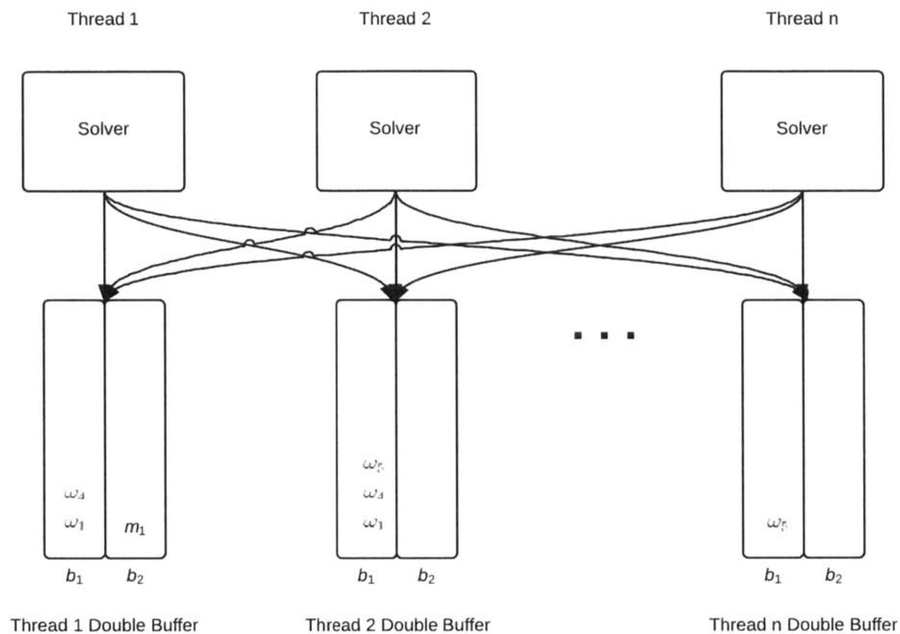


Figure 3-2: Each thread has a pointer to every other's double buffer. Other threads write to a write buffer while the owner reads from the read buffer. These buffers swap as the write buffer becomes full.

the buffer), it takes a lock out on that specific double buffer.

We also have the issue of some messages being never read by the owner of the double buffer. For example, consider the following execution with two threads:

1. Thread 1 writes message m to thread 2's write buffer, b_1. However, b_1 is not full and thus we do not swap buffers.

2. Thread 1 returns with some value.

Clearly, thread 2 will not receive m because thread 1 will send no more messages and b_1 will remain forever the write buffer. Therefore, we add one more operation to our system: FLUSHMESSAGES(INT *thread_id*). This operation checks to see if the read buffer of thread t_i is current empty, and if it is, swaps the read and write buffers. Now thread t_i is able to read the unseen messages without waiting until the write buffer is full. The FLUSHMESSAGES() procedure is useful when we want threads to read our messages immediately (usually during the correctness protocol). Note that this operation implies that readers are consistently reading messages, or a writer could be waiting for the read buffer to be empty indefinitely.

3.6.1 Clause and Message Format

Now we examine the structure of the messages passed in the communication channel. We have two types of messages, regular learned clauses, composed of LIT (literal) objects in Minisat and special messages used for the correctness protocol. To simplify the scheme, we encode the special messages as regular clauses. We noted that Minisat variables are integers from $0 \ldots n-1$, i.e. they are non-negative. Therefore, we use the first nine negative integers as our special messages, as shown in Figure 3-3. Furthermore, for both clauses and messages, we append a header to each message that corresponds to the size of the message so the reader knows how much of the buffer to read of the buffer for a particular message. This header is also encoded as a LIT object in Minisat.

1. **FOUND_SATISFYING_ASSIGNMENT** : CLAUSE(LIT(-1))
2. **STILL_FINISHED** : CLAUSE(LIT(-2), LIT(*seq_num*))
3. **FINISHED_ACK** : CLAUSE(LIT(-3), LIT(*seq_num*))
4. **NOT_FINISHED_ACK** : CLAUSE(LIT(-4))
5. **RETURN_SAT** : CLAUSE(LIT(-5))
6. **FOUND_UNSATISFIABLE** : CLAUSE(LIT(-6))
7. **UNSATISFIABLE** : CLAUSE(LIT(-7))
8. **UNSATISFIABLE_ACK** : CLAUSE(LIT(-8))
9. **RETURN_UNSAT** : CLAUSE(LIT(-9))

Figure 3-3: A description of the nine types of special messages. They are encoded using the same clause infrastructure built into Minisat.

3.7 Optimizations

3.7.1 Activity

A common heuristic for choosing the next decision variable is based on *activity.* That is, as we learn new clauses, modern SAT solvers bump the activity of the variables inside these clauses. This heuristic has proven to be effective in speeding up solve times. However, we have the idea of global variables, so we need to modify this heuristic slightly. For example, consider the following high-level scenario with two threads:

Thread 1 and thread 2 share some set of global variables, g. Thread 1 is incredibly constrained on the values of some subset of g, lets call this set s_1. Consequently, we will call the other subset of global variables s_2, which is $g - s_1$. This means that thread 1 has a relatively low set of values for s_1

that does not leave its instance unsatisfiable. On the other side, thread 2 has many sets of assignments of s_1, but s_2 is constrained. Ideally, we would want thread 1 to work on s_1 and thread 2 to work on s_2 without the other arbitrarily deciding values in the other set, which will disrupt the search process.

We can mimic this behavior using the activity mechanism built into Minisat. Whenever a thread receives a broadcasted clause, it decreases the activity of all the variables in the clause. The justification is the following: if a thread hears many of the same variables from another thread, it is working hard on discovering information about that particular variable, and arbitrarily setting it to a different value through decision will only hamper its progress. We found that this added heurstic increased performance significantly in nearly every instance in our test suite.

3.7.2 Sleeping and Empty Read Buffers

When a thread finishes its execution, unless it is the last thread to finish, it spins reading messages from other threads, waiting to terminate or to begin searching again. Thus, a thread is constantly reading the outgoing double buffers of each thread, which involves acquiring and unacuiring locks. As one might expect, this slows the communication channel immensely as the writer has difficulty acquiring the lock, which in turn delays the reader from actually hearing the message that will allow it to terminate! Therefore, we add a small sleep (10 microseconds) for each finished thread before attempting to read all messages again.

In a similar vein, we also keep a flag for each double buffer called *read_empty*,

which returns true if the read side of the double buffer is empty. When a thread attempts to read messages from other threads, it firsts checks the *read_empty* flag before acquiring the lock on the double buffer, reducing load on the double buffer when there is nothing to be read.

Both of these optimizations proved to give significant speedups.

Chapter 4

Benchmarks

4.1 Briefing

We now examine how our system fares versus the sequential Minisat, along with a state of the art portfolio SAT solver, PeneLoPe [AHJ+12]. PeneLoPe utilizes clause sharing along with a standard portfolio approach and was recently awarded second place in the parallel division of SAT Challenge 2012 (the winner's source code faced compability issues with our benchmark framework, and was therefore not used).

The hardware we used to run the tests was an MIT CSAIL server named sketch1 running Debian 2.6.26-29 and whose hardware specifications include roughly 7 GB of RAM and an Intel Core2 Quad CPU at 3 GHz.

The set of benchmarks we used are admittedly not the standard industrial benchmarks used in SAT competitions. Rather, they are from a program analysis tool called Sketch used in the Computer Assisted Programming (CAP)

group at CSAIL. In general, these problems have the "big but easy" property: they have a large amount of variables and clauses, but for modern SAT solvers, are solvable in a reasonable amount of time. Our justification in using these benchmarks is to demonstrate how a tailor-made partitioning algorithm can provide large speedups given the correct application. In this case, we used the natural partitioning algorithm due to the fact that the instances provided have a large degree of locality within the input files: the probability of a variable being in two clauses who are a large distance away from each other in the input file is relatively low. Thus, by splitting the file into simple, uniform, pieces, we can create a good partition with little overhead.

Finally, the procedure we used to produce the results shown in Table 4.1 are to run each SAT instance twenty times in an attempt to smooth out the randomness that can occur in SAT problems as well as parallel applications. Of these twenty trials we took the median values for RAM usage and time. Finally we summed these medians for each instance in the benchmark suite, arriving on the totals found in the above table. The Count column refers to the number of instances in that particular benchmark. PeneLoPe and ParallelSAT were both run on 2 threads.

4.2 Analysis

First and foremost, the results point to a speedup on nearly every instance compared to Minisat. The two set of benchmarks where ParallelSAT (and PeneLoPe) run slower, doublyLinkedList.sk and isolateRightmost.sk, are small instances where the overhead of parallelism is too much. In general, as the instances get larger and larger (as in the case of ConcreteRoleAffectationSer-

Benchmark	Count	Minisat		PeneLoPe		ParallelSAT	
		RAM Usage (MB)	Time (s)	RAM Usage (MB)	Time (s)	RAM Usage (MB)	Time (s)
ConcreteRoleAffectationService.sk	55	3394.36	532.57	17361.5	307.96	11472.19	331.52
Pollard.sk	2	8.43	0.06	292.0	0.19	191.68	0.08
SetTest.sk	7	37.88	0.28	1022.0	0.79	663.71	0.35
auction.sk	36	3000.93	373.47	13507.5	487.21	7887.53	100.1
compress.sk	18	100.44	1.08	2628.0	2.76	1742.24	1.86
diagStencil.sk	49	2450.05	33.97	12339.5	68.24	8026.23	37.78
diagStencilClean.sk	43	2069.22	27.42	10723.0	65.63	6801.4	31.9
doublyLinkedList.sk	7	14.68	0.05	1022.0	0.11	647.01	0.06
enqueueSeqSK.sk	8	20.44	0.14	1168.0	0.24	743.32	0.12
isolateRightmost.sk	4	10.13	0.03	584.0	0.1	371.72	0.05
jburnim_morton.sk	11	104.55	1.11	1606.0	2.71	1129.16	2.75
karatsuba.sk	6	21.86	0.26	876.0	0.47	560.61	0.22
listReverse.sk	5	20.7	0.3	730.0	0.64	477.31	0.24

Table 4.1: Table of benchmarks for SAT problems provided by Sketch for two threads.

vice.sk and auction.sk), both ParallelSAT and PeneLoPe give large speedups.

The other main result is the RAM usage between the three solvers. The sequential solver, Minisat, of course utilizes the least amount of RAM. The portfolio SAT solver, on the other hand, utilizes at least twice as much RAM as the sequential solver because it must copy all of the original clauses along with the overhead of parallelism. While ParallelSAT also incurs a memory overhead due to the extra parallelism of locking and clause exchange, it is considerably less than PeneLoPe. To see why this is true, we recall that by partitioning the instance's clauses, we are not required to copy the entire clause set into multiple solvers like the portfolio approach. This RAM difference between the two parallel solvers will grow linearly as the number of threads increase, as the portfolio approach must copy the clause set n times, where n is the number of threads.

One interesting metric of SAT solvers is the number of unit propagations in the search process. As mentioned above, most of the time of in the search process is spent propagating unit clauses, so the number of propagations is a

Benchmark	Count	Minisat Propagations	ParallelSAT Propagatations
ConcreteRoleAffectationService.sk	55	2777300343.0	420645632.0
Pollard.sk	2	140010.0	430192.0
SetTest.sk	7	703612.0	541096.0
auction.sk	36	1506597911.0	643000835.0
compress.sk	18	3592054.0	2770737.0
diagStencil.sk	49	71006295.0	33420183.0
diagStencilClean.sk	43	53883492.0	41390472.0
doublyLinkedList.sk	7	167020.0	114878.0
enqueueSeqSK.sk	8	673308.0	238847.0
isolateRightmost.sk	4	49140.0	111365.0
jburnim_morton.sk	11	3059776.0	3578750.0
karatsuba.sk	6	1548812.0	832792.0
listReverse.sk	5	1750041.0	1442374.0

Table 4.2: Propagations per benchmark for two threads. Note that this is the sum of the propgations for both threads.

convenient hardware independent metric between SAT solvers. The results for the total number of propagations with two threads can be found in Table 4.2.

We note that in most cases (especially larger SAT instances), ParallelSAT performs many less propagations than Minisat. This decrease in propagations suggests that ParallelSAT is not mainly getting its speedups due to the added parallelism, but rather the new approach of splitting the clause set into pieces and performing search with this partitioning in mind. Thus, this approach can be used in a sequential setting as well.

We also examined the cumulative distribution function (CDF) of propagations on particular SAT instances. SAT search is an inherently random procedure, so the CDF provides a helpful view of how the solver behaves over many runs. We discovered that ParallelSAT behaves more efficiently compared to Minisat as the instances grow larger and larger. We show a case study of five instances of the auction.sk benchmarks. We note that as the numbers of the instance

increase in our benchmarks, the number of variables and clauses increase (i.e, sat_SYN_12.cnf will be larger than sat_SYN_3.cnf).

As Figure 4-1 through Figure 4-5 indicate, Minisat becomes more and more random as the instances grow larger, while ParallelSAT stays relatively less random and becomes more efficient.

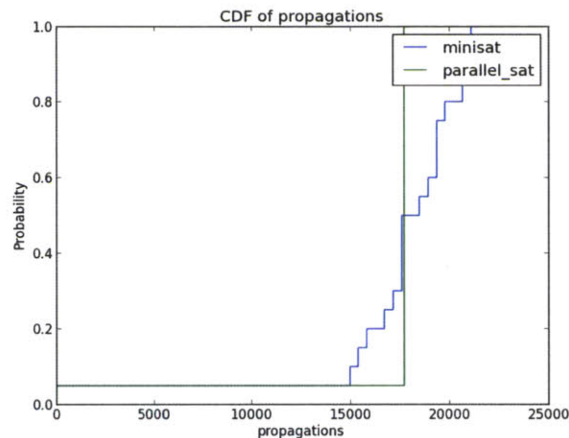


Figure 4-1: CDF of propagations for the smallest instance, sat_SYN_1.cnf

Now let's proceed to the most important metric of SAT solvers, the CPU Time. ParallelSAT runs quicker than PeneLoPe on every instance except the ConcreteRoleAffectationService.sk suite. Meanwhile, Parallel SAT completes the auction.sk set of benchmarks in roughly one fifth the time of PeneLoPe. To analyze this large discrepency, we will do a brief case analysis. We examined the largest instance in each set whose parameters can be found in Table 4.3.

As we note in the above table, the instance in the auction.sk benchmark suite not only has less global variables, but also has roughly one fifth the ratio of

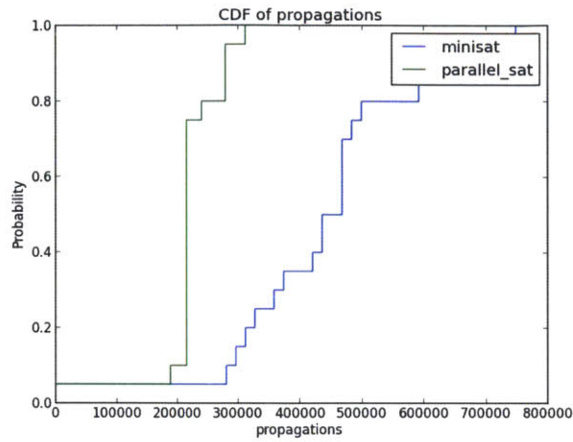


Figure 4-2: CDF of propagations for sat_SYN_9.cnf

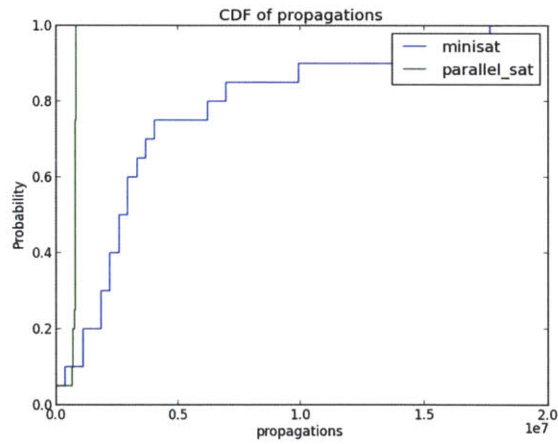


Figure 4-3: CDF of propagations for sat_SYN_17.cnf

global variables to the total number of variables. Having a low global variable count helps in many ways, including:

1. Reducing the amount of times a thread is forced to backtrack due to a broadcasted clause.

2. Reducing the traffic on the messaging system.

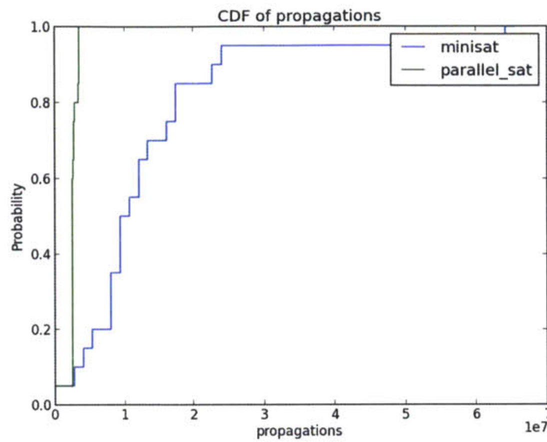


Figure 4-4: CDF of propagations for sat_SYN_25.cnf

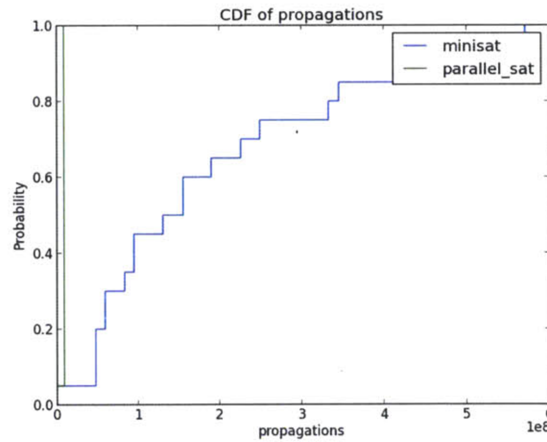


Figure 4-5: CDF of propagations for sat_SYN_33.cnf

3. Reducing the number of times a thread is forced to acquire a lock before setting a global variable.

Certainly there are other factors at play, but we have found that the global variable count and subsequent global count total number of variables ratio is one of the most consistent indicators of how well ParallelSAT will perform on

Suite	ConcreteRoleAffectationService.sk	auction.sk
Name	sat_SYN_135.cnf	sat_SYN_35.cnf
Variables	479973	1564422
Clauses	1834125	7994473
Global Variables	2352	1273
Global / Total	.0049	.0008

Table 4.3: Comparison of two particular SAT instances with two threads using the natural partitioning algorithm

an unknown SAT problem.

To be fair, these instances have the nice property that there are a relatively small amount of common variables between the clauses, so the partitions are generally very good. However, these benchmarks show that this approach can be very effective given the right class of problems.

4.3 Scaling

Scaling this approach to four threads and beyond is one the main weaknesses of the current iteration of the algorithm. As a case study, we examine one set of benchmarks, diagStencil.sk, comparing two threads versus four threads. The results for CPU time are shown in Figure 4-6.

Although omitted, the trend exacerbates itself as the number of threads grows even larger. This increase in CPU time can be attributed to the increase in global variables, as shown in Figure 4-7.

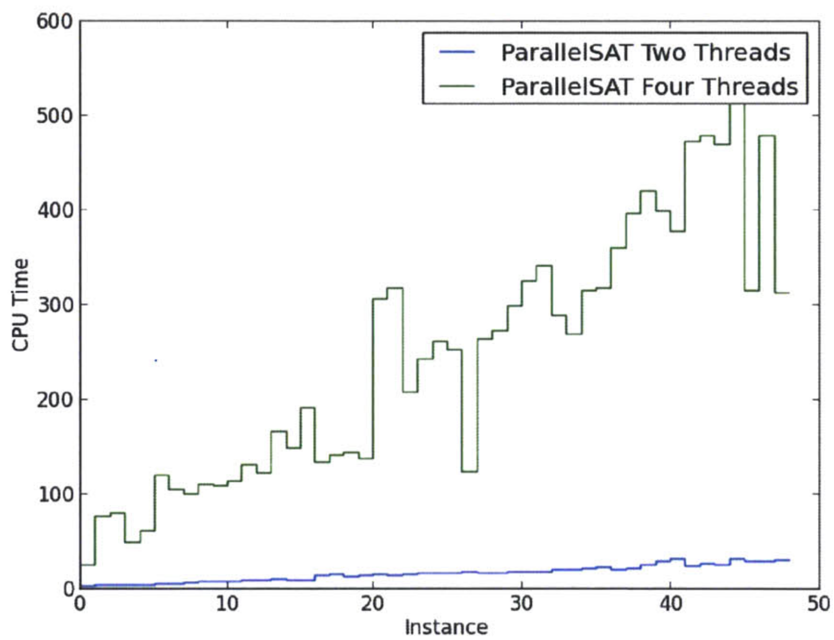


Figure 4-6: When the algorithm scales to more threads, the performance drops considerably. This can be attributed to a larger number of global variables which requires more interprocess communication.

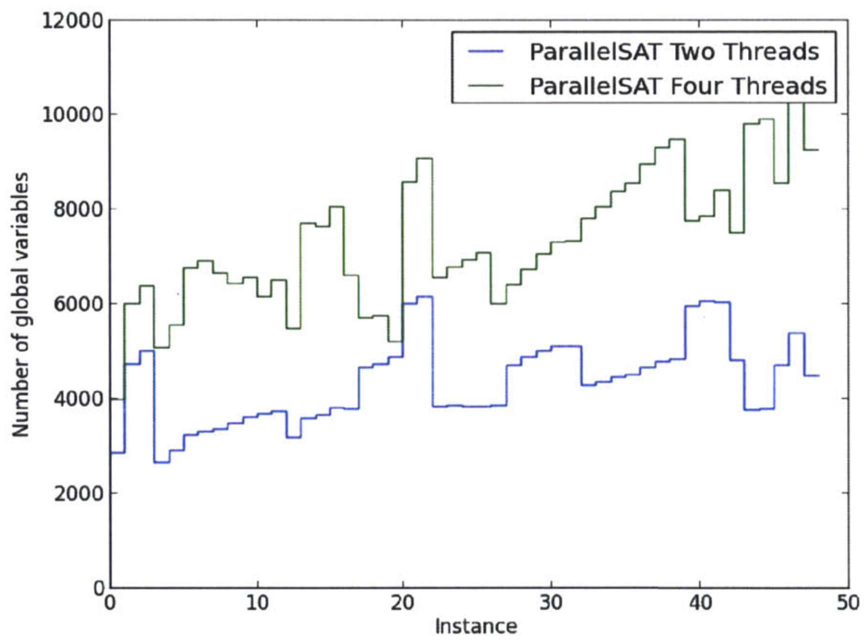


Figure 4-7: The number of globals grows considerably with more threads.

Chapter 5

Future Work and Conclusion

5.1 Future Work

As mentioned in the Scaling section, the main weakness of this algorithm is scaling to a greater number of cores. Extensions of this algorithms should focus on mitigating the problems that are associated wtih more threads, including greater numbers of global variables and more communication between threads.

One idea is to modify the the design decision to pick global variables first in the decision module. However, this would greatly increase the difficulty in producing global conflict clauses suitable for broadcasting. On the other hand, there could be great benefit in selecting some variables before others, as activity heuristics have shown us.

Another idea to explore is broadcasting clauses that contain local variables. You could imagine a situation where a global variable is constrained by a small set of local variables. To convey information about this global variable

to other threads, the thread must construct large clauses consisting of only global variables, which could be much smaller and simpler with local variables added. This would force other threads to be accomodating of new variables into their system, but could boost performance in the right context.

Lastly, a bottleneck for the general SAT case (with low locality in the input file) is the partitioner. Therefore, new approaches to partitioning (even for more specific applications like the high locality case), would be helpful.

5.2 Conclusion

In conclusion, we have presented a new parallel algorithm for solving SAT problems using the idea of partitiong the clause set into seperate pieces. We have shown how, given the right application, this approach can give significant speedups while simultaneously using less RAM than a world-class parallel portfolio SAT solver. While the current algorithm does not scale particularly well with added cores, this approach can be used alongside the portfolio approach in a computing cluster for a potentially faster and certainly more space efficient solver than running the portfolio solver alone on all cores.

Bibliography

[AHJ+12] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 200–213, Berlin, Heidelberg, 2012. Springer-Verlag.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[ES03] Niklas Een and Niklas Srensson. An Extensible SAT-solver [ver 1.2], 2003.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[JS97] Roberto Bayardo Jr and Robert C. Schrag. Using csp look-back techniques to solve real-world sat instances. pages 203–208. AAAI Press, 1997.

[Kar] George Karypis. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices version 5.1.0.

[KK95] George Karypis and Vipin Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001.

[MS08] Joao Marques-Silva. Practical Applications of Boolean Satisfiability. In *In Workshop on Discrete Event Systems (WODES)*. IEEE Press, 2008.

[MSS96] Joao P. Marques-Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability, 1996.

[XHHLb] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-brown. Satzilla: Portfolio-based algorithm selection for sat.

[ZMM01] Lintao Zhang, Conor F. Madigan, and Matthew H. Moskewicz. Efficient conflict driven learning in a boolean satisfiability solver. In *In ICCAD*, pages 279–285, 2001.