UNIVERSITY OF CALIFORNIA
Santa Barbara

GridSAT: A Distributed Large Scale Satisfiability Solver for the Computational Grid

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Wahid Chrabakh

Committee in Charge:

Professor Rich Wolski, Chair

Professor Kevin Almeroth

Professor Amr El Abbadi

September 2006

The Dissertation of
Wahid Chrabakh is approved:

Professor Kevin Almeroth

Professor Amr El Abbadi

Professor Rich Wolski, Committee Chairperson

June 2006

GridSAT: A Distributed Large Scale Satisfiability Solver for the Computational Grid

To my parents, my wife and the beautiful

trio Weam, Hazem and Raneem.

Acknowledgements

This PhD dissertation was not possible with the support and guidance of many people. First, I would like to thank my advisor Rich Wolski for his confidence in me. By sharing his vision and deep knowledge this research was made more encompassing. I must also thank the other members of my committee: Kevin Almeroth and Amr El Abbadi for their comments and probing questions which helped shape the focus of this work.

During my PhD career I met and interacted with many people who influenced in some way this thesis and how I approach research in general. Also I would like to thank all past and present members of the Mayhem lab for their friendship. I especially thank Graziano Obertelli for his methodical effort in maintaining the machines I used for many of my experiments.

Most importantly, I would like to thank my mother and father for their support. This work would not have been completed without my wife Hasna Bellagha. Hasna, your patience, love and companionship were indispensable during every step. You were always by my side cheering me on. I thank you deeply and sincerely. Finally, I am grateful for my three children Weam, Hazem and Raneem. They have provided me with the most memorable and precious part of my life. Their love and affection are overwhelming and were pivotal in the most trying of times.

Curriculum Vitæ

Wahid Chrabakh

Education

2006	Ph.D. in Computer Science, University of California, Santa Barbara
2000	Master of science in Computer Science, University of Tennessee, Knoxville
1994	Bachelor of Science in Electrical Engineering, University of Tennessee, Knoxville (Summa Cum Laude)

Experience

1997–2000	Lead Software Engineer, The Learning Company, Knoxville, TN
1995–1997	Lead Software Engineer, TELNET, Tunis, Tunisia
1994–1995	Software Engineer, ALCATEL Corporation,Tunis,Tunisia

Selected Publications

Chrabakh, W. and Wolski, R., "Solving 'hard' satisfiability problems using GridSAT", Global Grid Forum 14, June 27-30, 2004, Chicago IL.

Chrabakh, W. and Wolski R., “GridSAT Portal: A Grid Portal for Solving Satisfiability Problems On a Computational Grid”,Global Grid Forum 14, June 27-30, 2004, Chicago IL.

Chrabakh, W. and Wolski, R., “GridSAT: A Chaff-based Distributed SAT Solver for the Grid”, Proceedings of supercomputing, Phoenix, AZ, November, 2003.

Chrabakh, W. and Wolski, R., “GridSAT Portal: A Grid Web-based Portal for Solving Satisfiability Problems Using National Cyberinfrastrcture”, Journal of Concurrency and Computation: Practice and Experience, Published by John Wiley & Sons 2006 [*to appear*].

Chrabakh, W. and Wolski, R., “GridSAT: A System for Solving Satisfiability Problems Using a Computational Grid”, Journal of Parallel Applications, Published by Elsevier, ISSN: 0167-8191 Imprint: NORTH-HOLLAND 2006 [*to appear*].

Chrabakh, W. and Wolski, R., “GridSAT: Design and Implementation of a Computational Grid Application”, Journal of Grid Computing, Publisher: Springer Netherland 2006 [*to appear*].

Experience

Chrabakh, W. and Wolski, R., "GrADSAT: A Parallel SAT Solver for the Grid", University of California, Santa Barbara Computer Science Technical Report Number 2003-05, February, 2003.

Invited Professional Talks

"The Making of a 'true' Computational Grid Application", SIAM Conference on Parallel Processing for Scientific Computing, Software Development, February 2006.

"Solving 'hard' satisfiability problems using GridSAT", Global Grid Forum 14 (GGF14), Grid Applications: from Early Adopters to Mainstream Users, June 2005.

"GridSAT Portal: A Grid Portal for Solving Satisfiability Problems On a Computational Grid", Global Grid Forum 14 (GGF14), Science Gateways: Common Community Interfaces to Grid Resources, June 2005.

"GridSAT: A 'Chaff' Based Satisfiability Solver", SuperComputing Conference, Applications, November 2003.

Demonstrations

SuperComputing Conference 2004,	Demonstration in conjunction with San Diego Supercomputing Center (SDSC).
SuperComputing Conference 2003,	Flagship demonstration as a leading application using SDSC resources (four times).
SuperComputing Conference 2002,	Demonstration as a component application of the Grid Application Development Software (GrADS) project.

Software and Tools

GridSAT Portal:	Available at http://orca.cs.ucsb.edu/sat_portal. The GridSAT Portal is a simple web interface for solving Satisfiability problems using supercomputing cyber-infrastructure. The portal uses GridSAT to automatically solve user supplied Satisfiability problems.
GridSAT System:	The GridSAT System is portable and can be deployed on any set of computational resources.

Technical Referee

SuperComputing conference 2002, 2003, 2004.

High Performance Distributed Computing 2002, 2003, 2004.

Journal of Parallel and Distributed Computing.

International Conference High-performance Distributed Computing.

International Conference on Parallel and Distributed Systems.

Abstract

GridSAT: A Distributed Large Scale Satisfiability Solver for the Computational Grid

Wahid Chrabakh

Grid Computing is an emerging field in computer science. Research in this area aims at aggregating distributed, heterogenous and federated resources and make it available to Grid applications. In the past two types of applications have been deployed with varying degrees of success. The first type of applications is embarrassingly parallel (a bag of independent tasks). This category adapts well to a computational grid environment. The second category of applications includes mainly scientific code which is tightly coupled in nature. This type of applications is very hard to deploy in a grid environment.

In this thesis we present GridSAT, a new grid application. GridSAT is a distributed complete boolean satisfiability solver based on the sequential solver Chaff [70]. In addition to its theoretical significance, the satisfiability problem has numerous practical applications. SAT solvers are used in many engineering and scientific fields including circuit design and model checking.

GridSAT is able to achieve new results by solving faster those problems that were previously solved by other solvers. Moreover, it was able to solve problems which were left unsolved by other solvers. GridSAT accomplishes these results by achieving two goals. The first is parallelizing the sequential solver in a manner which allows it to run efficiently on a large collection of resources. GridSAT also uses techniques to enable information sharing between the parallel components to avoid redundant work. The second goal is to design and implement the application so that it can adapt to the dynamic conditions of a computational grid environment. The techniques and design used to realize GridSAT can be deployed with other application to achieve new results.

In addition, we show how multiple GridSAT instances can cooperate to run efficiently on a common set of resources without explicit synchronization. These experiments represent realistic scenarios where many grid applications share a common resource pool.

We have also developed a web portal which accepts problem instances through a standard web browser and returns status and results while shielding users from complexities of running the application manually.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many practical applications require solutions to arbitrarily complex boolean constraint problems. Also this type of problems is very important from a theoretical perspective. In this dissertation we present a new application *GridSAT* capable of using a diverse set of computational resources to solve previously unsolved boolean constraint problems. The application is capable of dynamically adapting its resource usage based on the computational environment. By combining existing federated and geographically distributed computational infrastructure the presented application makes it possible to solve problems that were up till now out of reach for existing solvers; thus achieving new domain science results.

1.1 Computational Grid Computing and Related Challenges

The use of computational resources has experienced many epochs with every new technological progress in both computation units and networking. In the past decade, there was an accelerated proliferation of computational power in the form of more powerful workstations, small scale clusters and supercomputers. These computational resources are ubiquitously connected using faster more reliable networks. If used collectively this large set of interconnected computational resources presents enormous compute potential. Achieving this goal makes it possible to satisfy the ever growing need for compute power by many applications in the area of science and engineering. It will also enable the deployment of a new generation of applications. For the past decade, many research efforts have been directed towards realizing this potential. One such area of research that aims to enable the simple use of large sets of federated heterogenous resources is *Computational Grid Computing.*

Grid computing [43, 45, 68] is based on the notion of software enabled mediation to realize its grand vision. This vision is usually expressed by using the electrical grid as a metaphor for the computational grid. Just like the electrical grid provides electrical power to electrical appliances, the computational grid enables applications to consume computational power. This metaphor aims at imitating three characteristics of the electrical grid: heterogeneity, federation and geographical distribution.

First, electrical appliances can consume electrical power no matter its provenance: fossil, solar, wind, etc. Similarly, computational applications should be able to use compute power regardless of the device. In a computational grid environment, resources can be heterogenous in hardware, operating system or other software tools and libraries a grid application may interact with. This aspect of a grid environment presents two challenges: variable performance and portability.

Traditional providers of large computational power present uniform predictable performance of the underlying resources from the point of view of the application. All computational nodes are usually identical in hardware and software. In contrast, components of a computational grid are diverse. Thus their performance are not identical and can in fact span a wide range. For example a hand-held device and a supercomputer can be part of the same grid. Also each resource's performance as observed by an application may change over time. This change occurs most often because of other programs concurrently running on the same resource. Other factors may also affect a particular resource such as malfunction, software and hardware upgrades and other administrative decisions. Therefore, a successful deployment over these resources should preserve high performance in spite of these global and time dependent variations. A grid application has to use adaptive scheduling to enable resource usage patterns that maximize its overall performance.

The other challenge is portability. Portability across all levels of heterogeneity is very important in order to reduce development cost and debugging complexity. Portability can be viewed in two ways. First, portable code allows the application modules to execute on various platforms using standard compilers and libraries. This is achieved to varying degrees using standard programming languages such as C or Java. Second, portability can be viewed as the ability of the application to interact with remote services as well as other applications. This type of portability can be achieved using standard interfaces that allow for uniform interaction regardless of the implementation.

Second, the electrical grid delivers electricity from various geographical locations depending on demand. Compute power should also be harnessed from geographically distributed sites depending on their availability and computational needs of applications. A distributed resource pool presents additional communication overhead for the application. Overall the network delay is increased and the network bandwidth is reduced. In addition, since these network links are shared there will be variability over time in the observed characteristics. Enabling applications that can tolerate lower performance and are resilient to dynamic variations is a complex task. Such network characteristics further complicate scheduling policies for computational grid applications.

Third, electrical power is generated by various independent companies. In the same way, different sites with local administrative control can contribute computational power to a computational grid. In a computational grid local administrators have complete control over their resources. Resources can be updated, upgraded or even removed without warning outside users. Also new resources can be added without notice. As such, an application is not guaranteed a quality of service or even the continued use of a particular resource. This puts yet another burden on grid applications because they have to deal gracefully with resources leaving and joining their execution environment. Thus these applications need mechanisms that allow them to discover and update the status of all potential resources. These mechanisms need to be active during the entire duration of an execution and not just at the beginning when applications are instantiated. This especially important because grid applications usually require long runtimes.

The metaphor of the electrical grid used to describe the computational grid characterizes the desired view point of the end-user. This metaphor, however, breaks in various ways from a developers perspective. An electrical appliance is geographically located in a single place while the electricity travels to it from different generators. In contrast, a computer application has to send its components to the various resources where they can consume computational power. Also, all electricity is the same independent of its origin but computer applications might have resource preferences. For example, an application might only run on certain hardware or require specific software support. In addition, computer applications might prefer certain collection of resources connected by special networks to achieve desired performance level.

By promising to mitigate the challenges presented by the main features of a computational grid environment: heterogeneity, geographical distribution and federation, the computational grid promises two inter-related goals: ease of use like the electrical grid and enabling advances in science by achieving new scientific results.

1.2 Solving Boolean Satisfiability Problems

One of the most intriguing problems in science are those that are simple enough to explain even to non-experts but are very hard even for the brightest specialists to find a solution for. Such is the boolean satisfiability problem. Stated simply, this problem answers the following question:

> Given a set of variables which can be either *true* or *false* and logical formula using logical *AND* and *OR*, is there an assignment of values to variables so that the entire formula is true.

Boolean SAT is a critical problem to solve, but it is so complex, it requires extensive computational capability. This problem is the first problem to be proven NP-Complete [30] and is therefore computationally intensive. There are several engineering and scientific domains that require the solution of domain-specific instances of satisfiability. Satisfiability is especially important in the area of Electronic Design Automation (EDA). EDA encompasses a variety of problems such as circuit design [96], Field-Programmable Gate Arrays (FPGA) detailed routing [71], combinational equivalence checking [60, 80] and, automatic test and pattern generation [62]. Other disciplines where satisfiability solvers are used include scheduling [11], model checking [20], security [17], Artificial Intelligence [59] and software verification [56]. Because of the importance of this problem in many practical fields and theory, there has been a plethora of research aimed at generating more and more efficient solvers.

The problems generated in practice use thousands and even millions of variables [123]. Solvers must check, speculatively, different possible truth assignments creating a search space the size of which is an exponential function of the number of variables. Thus the size of the search space for such problems is very large. Actually, it is extremely large since it dwarfs the number of particles in the universe, that is estimated to be between 2^{78} and 2^{200}. Therefore, a naive case-by-case investigation would take a prohibitively long duration. Many more efficient algorithms have been developed. The most commonly used algorithms are sequential. It tackles a given problem by investigating its search space while using optimizations that allow them to prune the search space and avoid redundant work. This optimization termed *learning* was shown to very successful and is now adopted into most solvers. As the algorithm progresses, learning is used to gain more knowledge about the problem at hand. This knowledge is stored in a large database. As part of its progress the algorithm frequently accesses and updates the knowledge database. Thus the database is kept memory resident for efficient access.

Another approach to speeding-up sequential solvers is parallelization. Parallelization is usually applied by splitting the original problem to many subproblems. Combining parallelization with learning presents some problems since the knowledge base produced by the learning process requires frequent updates. From this perspective, satisfiability algorithms and their implementations are hard to parallelize in a computational grid environment since a simple distribution of the knowledge database will result in significant slow down. The main cause is the high communication and synchronization overhead.

1.3 Thesis Statement

A new approach to SAT application design that exploits distributed and parallel resources may lead to a new solver capability. This thesis presents a novel distributed SAT solver designed to adaptively execute in a dynamic computational environment and demonstrates that:

> "It is possible to enable a more powerful satisfiability solver in the form of a 'true' grid application that is capable of harnessing the computational power of large collections of dynamic resources while enabling new domain results."

The first step in proving this statement is by implementing a satisfiability solver capable of using multiple resources while using the most efficient known optimizations such as learning. Such a solver should provide a significant speed up over existing solvers. Since there is no theoretical basis for ensuring that such a solver exists, the distributed solver is to be evaluated empirically and should show speed-up in the majority of cases. We intend to use large collections of computational resources to attempt to solve problems that are left unsolved by existing solvers. By answering conclusively whether those problems are satisfiable or not we intend to prove that the new solver provides new capability.

The solver should also be able to adapt to variations in the SAT instance and the computational environment. Some SAT instances are trivial while others are complicated. The solver should use as much resources as needed to solve a particular problems. For example, trivial problems should use few resources while more complex ones may use all available resources. The solver should be able to selectively run on those resources that are more probable to advance the solution of the problem at hand. It should not use those resources that might appear to be operational but provide no additional actual help to the solving process. The solver should continually check when resource fall in to this category.

Another measure of the success of the application is its ability to gracefully share resources and adapt in the presence of other grid application. The application should be able to cooperatively run and make progress while sharing resources with other distributed applications. Deploying many instances of application simultaneously would validate there ability to adapt in the presence of other applications. Also we can determine how sharing resources effects their solving capabilities.

1.4 Existing Satisfiability Solvers

The boolean satisfiability problem is very famous and is also known simply as the SAT problem. The SAT problem is at the core of many computationally intractable problems. It was the first problem to be proven NP-complete by Cook [30] in 1971. Even before this proof was published, researchers were looking as early as 1962 [36] for efficient algorithms to solve this problem because it was identified as central to many problems is science and engineering. Because it is NP-complete, it is unlikely that any algorithm has a fast worst-case time behavior. Moreover, there is currently no known method to predict accurately the runtime of a particular algorithm for a given problem. In spite of all of this, research has continued unabated into finding gradually more effective and clever methods to solve SAT problems of practical importance.

Over the past four decades there was tremendous progress in generating more and more efficient solvers. At the same time new practical applications for satisfiability have been introduced. Currently, SAT solvers are being used in many fields of science and engineering such as automated reasoning, computer aided design, computer circuit design and computer network design [123].

In spite of this progress, there are many problems that are now considered out of reach for current sequential solvers. In fact, a number of benchmark problems during the SAT competition [89] are left unsolved. Also, there are many theoretical problems that have so far been left unproven because of their difficulty can potentially be solved after being expressed as a SAT problem.

Current solvers run as sequential code and their performance is sensitive to both CPU speed and the size of memory on the host machine. One approach to extending the capability of existing solvers is to increase the computational power that these solvers can use simultaneously. The fastest sequential solvers only run on a single machine because of frequent memory updates. One possible method is to use even larger machines with larger memory. Since such machines are in heavy demand it will be unlikely that all potential users will have access to such machines. Also, the cost of these machines will be out of reach for many users.

Another way to increase computational power is to allow the simultaneous use of many machines. Many research efforts have produced parallel solvers [58, 97, 41, 21]. These efforts were limited in two aspects. First the cooperation between the parallel components was restricted. Second, the types of resources used were constrained to a small set of locally connected homogenous workstations. Our approach explores cooperation between the parallel components to increase the overall efficiency of the solver. Since the basic algorithm learns new information throughout its execution, our solver shares this information so that other components do not have to waste resource relearning the same piece of information. Our goal is also to make it possible to incorporate any new advances in sequential solvers with minimal effort. Thus our solver allows new agents to be executed on other computational resources and be integrated almost effortlessly.

Also we aim to use all and any computational resources available. This strategy accomplishes two goals. First, users can start solving there own problems immediately. This has the potential to solve those problems that are *easy* immediately without unnecessary delay as the user may wait for a long time before new resources are made available to him. Second, the user can make use of all potential computational resources no matter where they originate from. This assures the user that he is provided with the best chance to solve a given problem.

In the next section, we discuss how our application relates to the current state of grid computing which made it its core mission to aggregate computational power and make it easily available for users.

1.5 Current State of Application Development in Computational Grids

Grid computing research has produced many advances to realize its vision of seamless aggregation of computational power, data centers and scientific instruments to enable new scientific results and change the way in which computing centers and organizations interact.

The results are middleware and other tools that provide basic application services. Figure 1.1 shows a generic view of a grid application and how it makes use of current grid services and components. Some of the grid components may become part of the application such as libraries. At the same time, a grid application may also interacts with other grid services that are totally disjoint from the application.

The infrastructure created by computational grids must be usable by potential *grid applications.* These application are the lifeblood of future success of grid computing. In the past, two types of applications have been deployed on the grid. The first type consists of embarrassingly parallel applications. These are applications that can be divided into a –usually large– set of totally disjoint tasks. As such they are suitable for the computational grid because they can easily cope with the dynamic nature of the computational environment. The second type of applications does not cope so well with highly variable performance and availability of resources. These are applications that require frequent synchronization and are sensitive to variations in connectivity and discrepancy in performance between different hosts.

The application we present in this thesis provides a new type of applications that can dynamically adjust their communication and computation needs. It represents a middle ground between both extremes represented by the two types of application mentioned above. In addition to being malleable, GridSAT can overlap computation and communication. As such, our application presents a new type of application distinct from those previously deployed in the context of computational grids. There are many other applications that share these characteristics and can be deployed in a similar fashion.

Like many systems, the design and deployment of computational grid services and resources will be influenced by those applications that will use them. By providing a diverse set of applications the grid infrastructure is influenced to take into consideration a wider scope of actual application features.

Moreover, GridSAT was designed to take advantage of existing grid services and tools without being tied to anyone in particular. GridSAT is written from first principals and shows another example of how many grid services are integrated into new applications. Also GridSAT is capable of tuning its footprint on host machine so as to be cooperative with other applications. This programming methodology allows for efficient use of resources and avoids overloading them.

Finally, the unique design of GridSAT made for a simple portal design that does not get clobbered by explicit resource management.

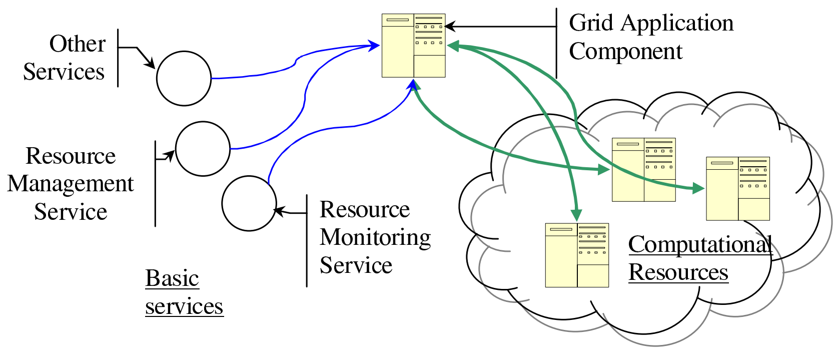

Other Services
Grid Application Component
Resource Management Service
Resource Monitoring Service
Basic services
Computational Resources

Figure 1.1: Simplified view of a generic grid application.

1.6 Research Plan

In order to answer the thesis question in Section 1.3, we plan to use as basis for our research the best available sequential solver [120]. We take two separate but complementary steps in order to realize the target application. First, we parallelize the solver and introduce additional optimizations for sharing information between the parallel components. Second, the parallel components are modified so that they can execute in a computational grid environment.

Finally, the application is experimentally evaluated to show its performance and stability. We conduct four experimental sets.

- The first set of experiments evaluates the many options and parameters the parallel solver could employ. The results help us identify which strategy is best and how significant the improvement is.

- The second set of experiments compares the parallel solver to the original sequential solver in order to evaluate when using more resource helps improve solver speed.

- The third set of experiments proves that GridSAT can employ a large set of diverse resources for extended time periods to demonstrate its stability and scalability while tackling and solving some previously unsolved problems.

- The final set of experiments shows how many GridSAT instances can share a common set of resources in a cooperative environment.

The remainder of this dissertation is organized as follows. Chapter 2 introduces the satisfiability problem and sequential SAT solvers, while Chapter 3 describes the parallel algorithm used and the optimization employed in a parallel setting. Chapter 4 presents the programming methodology and architecture used in our application. Different experimental results evaluating our solver and the techniques used are discussed in Chapter 5. In Chapter 6, we present the portal developed to make the application available through a simple web based interface. Finally we summarize related work in Chapter 7 and conclude in Chapter 8.

Chapter 2

Background: Sequential Satisfiability Solvers

In this chapter we introduce the current state of the art for sequential satisfiability solvers. We also fill in the details and important insight into some of the techniques used in these solvers. The problem of propositional boolean satisfiability belongs to a larger group of problems. These problems are classified as *constraint satisfaction problems* [84]. In the rest of this document we will the refer to the boolean satisfiability problem simply as the satisfiability problem. This problem was first proven to be NP-complete by Cook [30] in 1971. However, the first algorithm proposed to solve this problem was more than a decade earlier, precisely in 1960 by Davis and Putnam [37].

In the past four decades a lot of research effort has been put towards producing more efficient and faster solvers. In general, SAT solvers can be classified into two types: *complete* and *incomplete*. Complete solvers are guaranteed to show that a specific SAT instance either has at least one solution or no solution. The solver will only halt after proving one of these results. Incomplete solvers, however, can generally produce faster solutions for satisfiable problems (i.e. a solution exists) because of the heuristics they employ. These solvers have an important drawback since they do not terminate for those instances which are not satisfiable. In the rest of this chapter we will only consider complete solvers.

We chose as the basis for our research the most powerful sequential solver available zChaff [70]. zChaff is an implementation of Chaff [120] by *L. Zhang* from Princeton. Chaff is a design of SAT solver with certain effective optimizations and heuristics. Using the best available solver will allow us to leverage the efficient techniques used by this solver. It also allows us to study how well these techniques perform when deployed in new computational environments. In the next section we start by defining the SAT problem and describing the most basic algorithm that is the basis for most modern solvers including Chaff. Later we describe additional optimizations before we present optimizations and heuristics specific to zChaff.

2.1 Definition of the Satisfiability Problem

This section defines the satisfiability problem and related terminology. We start by stating the satisfiability problem. Surprisingly, even though the satisfiability problem is associated with complexity, it can be formulated briefly in very few words:

> Given a boolean formula using a set of variables. Find an assignment of values to variables such that the formula evaluates to *true*.

More formally, a satisfiability instance is defined as follows:

- Given a set of variables $\{V_i | 1 <= i <= N\}$, where N is the number of variables.
- The domain of each variable is {*false,true*} or equivalently {*0,1*}.
- A literal is defined as the instantiation of a variable V or its complement $\sim V$.
- The satisfiability formula is a finite boolean expression consisting of a set of literals and boolean operators: conjunction (AND, . or $\wedge$)and injunction(OR, + or $\vee$). In the rest of this document we will use + and . notations (the latter is sometimes implicit in the notation).

The boolean satisfiability problem differs from other Constraint Satisfaction Problems (CSP) in two aspects. First, the variables in a SAT problem are restricted to the boolean domain while in a CSP the domain of variables can include many values and can even be continuous. Second, the constraints in a CSP are more complex and varied and are not usually limited to a single logical expression. A SAT formula is *satisfiable* if a set of assignments of values to variables makes the formula evaluate to *true*. If no such assignments exists the formula is called *unsatisfiable*. An assignment is called *complete* if all variables used in the original problem are assigned values, otherwise the assignment is called *incomplete* or *partial*.

Usually the SAT formula is expressed in *Conjunctive Normal* Form (CNF). Requiring the formula to be in CNF does not limit the set of formulae that can be used because there exists a polynomial time transformation algorithm to transform any boolean formula to an equivalent one in CNF [35]. This transformation might come at the cost of additional variables added to the new formula.

Formulas in CNF are represented as a conjunction of *clauses*. Each clause is an injunction of literals. The CNF is special in two ways. First, all clauses have to be true for the entire formula to be satisfied. Second, only one literal per clause needs to be true. Clauses can be called *satisfied*, *unsatisfied* or *undetermined* under a given assignment. In the first and second case the clause evaluates to true and false respectively. In the last case, the value of the clause cannot be determined based on the partial assignment under consideration.

For example the following formula is in CNF: $(V_1{+}{\sim}V_3)({\sim}V_2{+}{\sim}V_1)({\sim}V_2{+}V_4)$. The formula has three clauses and uses four variables. In this case the formula is satisfiable since setting $(V_1 = 1, V_2 = 0)$ make all clauses true and thus the entire formula evaluates to true. This is an incomplete assignment since V_3 and V_4 are not assigned values.

Usually there are trivial occurrences in the formula that could be checked by inspecting the formula once. For example, if a literal occurs more than once in a clause, then only one of the occurrences is kept. Also, if both literals from the same variable exist in the same clause, then that clause can be removed since it will always evaluate to true. Finally, clauses with single literals can immediately lead to variable assignment for the variable represented by that literal in a way that makes the clause true. Consequently, those clauses can be removed and the corresponding variable assignments are saved as part of a possible solution. For the rest of the discussion we assume that such trivialities have been addressed.

2.2 Sequential Algorithm

Our solver GridSAT is based on the sequential solver zChaff. zChaff is in turn based on Davis-Putnam-Logeman-Loveland (DPLL) [36]. We start by presenting this algorithm and related optimizations before we show the modifications and extensions needed to enable it to execute in a computational grid environment.

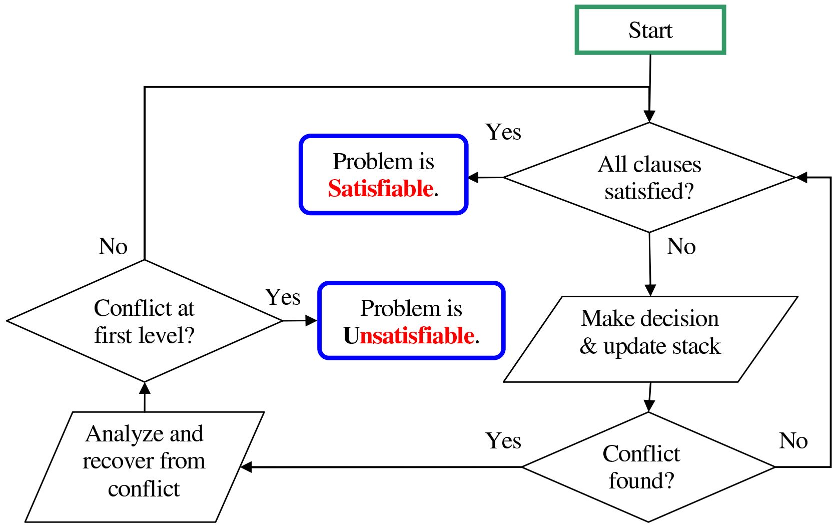


Figure 2.1: Flow chart for the DPLL algorithm

Figure 2.1 represents a block diagram of the basic algorithm which is the basis for most modern SAT solvers. The algorithm is based on tentatively assigning values to variables using some implementation dependent heuristics. These heuristics are designed to enhance the chances of yielding a solution quickly if one exists. However, these successive assignments most often result in a logical contradiction, also called a *conflict*. The conflict indicates that the current partial assignment cannot be extended to a satisfying one. Therefore, in order to maintain logical consistency, part of the partial assignment needs to be undone before the algorithm can continue. When a contradiction occurs, the algorithm backtracks and tries other decisions while avoiding repeating previous work. The process is repeated until a solution is encountered or the algorithm proves that no such solution exists.

The next section describes how the basic DPLL algorithm proceeds in more detail.

2.2.1 Backtrack Searching

The basic DPLL algorithm takes successive steps which assign values (i.e. *true* or *false*) to variables. Initially all variables are unassigned and marked as *unknown*. The speculative assignment of values to variables is called a *decision*. Each decision consists of the assignment of a boolean value to a single variable. A decision usually results in more clauses becoming satisfied (i.e. evaluate to *true*). These decisions are incremental as the algorithm adds a single new decision on every iteration. Each new decision can potentially cause one or more conflicts as some clauses become *false* because of the new variable assignment. The algorithm must resolve every existing conflict before making the next decision. Conflicts are resolved by undoing some of the previous variable assignments and possibly adding new ones. The decisions are stored in a stack to facilitate addition and removal of new and old decisions respectively.

In DPLL, each decision is given a unique decision level. After each decision the algorithm inspects the set of clauses in order to determine the status of the entire formula. If the formula can still be satisfied, the algorithm continues further by making new decisions. In case the formula cannot be satisfied because a clause is *false*, the algorithm simply *backtracks* by undoing the last assignment decision and trying the opposite value for that variable. This is accomplished in two steps. First the variable assignment corresponding to that decision is removed. Later, a new decision is added where the same variable is assigned the inverse value.

When a decision is inverted, the corresponding variable is marked as flipped. When the algorithm tries to flip a variable it checks if it has been flipped before. If the variable is not flipped, then the algorithm flips it at continues as usual. In case that variable has been tried both ways then that decision is removed. Now a new decision is at the top of the stack and is treated in a similar fashion. Thus the algorithm stops backtracking when a decision level is reached where the corresponding variable is not marked as flipped. This type of backtracking is called *chronological* because it tries to undo the most recent decision first. The process continues until a satisfying assignment is found or the decision stack is emptied. This happens when the algorithm backtracks in a way that results in removing the first decision in the stack.

The first decision level is reserved for a special class of variable assignments. This class contains those variables which have been assigned *final* values and these cannot be undone. When a variable is assigned a final value then the formula will only be satisfiable if that variable has that assigned value. For example, the variables whose values are deduced because of single literal clauses, as described in previous sections, are added to the first decision level. We can say that variables on this level have *known* values. There are other circumstances under which more variables are added to the first decision level as will be shown in section 2.3.

The DPLL algorithm with backtracking terminates when either a solution is encountered or it backtracks to the first decision level. In the former case, the problem is satisfiable and a solution is the partial or full assignment represented by the decision stack. In the latter case, when the DPLL algorithm reaches the first level then a conflict has occurred since both possible values of the decision variable at level two were tried without finding a satisfying solution. Therefore, the problem is determined to be unsatisfiable.

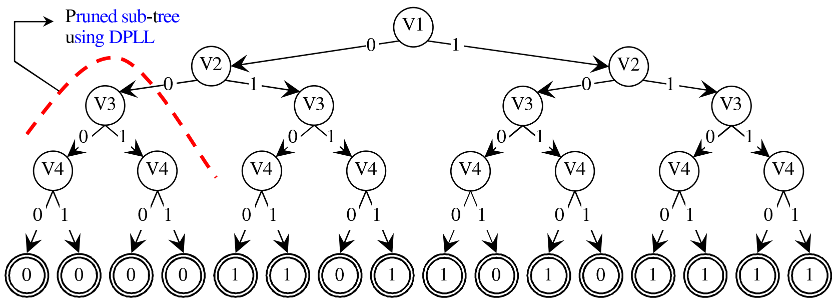


Figure 2.2: Illustration of Depth-First-Search vs the search used by the DPLL algorithm. The satisfiable paths have a leaf value of 1 (or *true*)

The DPLL algorithm is similar to a Depth-First-Search (DFS) in a binary tree representation of the search space. The tree in figure 2.2 shows the search space for the SAT formula $(V_1 + \sim V_3 + \sim V_4)(V_1 + V_2)(V_2 + \sim V_4)$. Each node in the tree represents a variable. Each node has two sub-trees one for each possible value of the variable. The leafs of the tree represent the value of the formula under the full assignment represented by the path from the leaf to the root. The total number of leafs is equal to 2^N – the size of the search space. A simple DFS visits the leafs in order and may visit all 2^N leafs before terminating.

The basic DLL algorithm, however, does not visit all the states because for each non-satisfying partial assignment it tries, it skips the rest of the subtree. The reason is that a partial assignment that falsifies one of the clauses simply cannot be extended to make a longer assignment which satisfies the formula. As shown in figure 2.2, the DPLL algorithm skips the highlighted subtree because it encounters the unsatisfying partial assignment ($V_1 = 0, V_2 = 0$). Thus the DPLL algorithm skips all other possible partial or complete assignments if they contain the current unsatisfying partial assignment. The algorithm, however, is still exponential in complexity.

It is important to note that the DPLL algorithm explores two search spaces. The first is the assignment search space that represents all the possible assignment of values to the set of variables. This is the formula search space which size is 2^N. The other search space is the *path search space* that represents the navigation space. The path search space consists of all possible orders of variable assignments that the algorithm may use to search for a solution. This space is ($N!\ 2^N$) which is much larger than the assignment search space. Referring back to figure 2.2, we have assumed for simple DFS that the variables are assigned values in a given order. Thus the number of possible paths is restricted to consisting of 2^N – number of all distinct paths from root to leaves. After each conflict, however, an algorithm such as DPLL can select any other variable for the next decision. For example, each node in the tree of figure 2.2 can be viewed as just one possible outcome of variable selection. Thus, at the root of the tree there will be N possible nodes. At the next level, each of the root nodes can select anyone of the remaining $N-1$ variables. The same procedure continues for later levels where the child nodes cannot reselect any variable previous selected in any of the parent nodes. The result is a representation of all possible selection paths which differ in the variable assignment order but share the same set of final states.

2.2.2 Boolean Constant Propagation

An optimization used by the DPLL algorithm is Boolean Constant Propagation (BCP). BCP improves performance because it automatically deduces new variable assignments based on the current partial assignment. Automatically augmenting the decision stack reduces the total number of decisions made by the algorithm since certain parts of the search space are pruned.

The BCP process inspects the clauses to find whether the values of other variables can be deduced as a results of previous variable assignments. After each new decision, the algorithm searches for *unit clauses*. These are clauses with a single unassigned literal and all other truth values false. In a unit clause, the last remaining literal must have the value *true* for the clause to be also *true*. When the algorithm encounters a unit clause, it sets the previously unknown literal to *true*.

When a literal is set to *true* because of a unit clause, this is called an *implication*. The corresponding variable is assigned the value that makes the literal *true* and is pushed onto the current decision level. Therefore, each new unit clause results in a new variable being assigned a truth value. The BCP process is repeated for each new implication which in turn may lead to the discovery of new unit clauses. As a result, more implications might be added in a cascade because of earlier implications. All these implications are added in sequence to the current decision level.

BCP, therefore, allows each decision level to contain a series of new implications in addition to the decision variable. Even though an implication is a direct result of the previous assignment, it is also predicated on some subset of the previous variable assignments. Since the decisions made by DPLL are speculative, propagating those decisions or later implications can result in a contradiction.

BCP is the most costly process in DPLL based solvers because it requires examination of the entire clause set. In addition, it is frequent because it is executed after every new variable assignment as a result of either a decision or an implication. There are many implementations of BCP described in the literature [123]. According to some studies, BCP accounts for up to 90% of the runtime of SAT solvers [70]. In section 2.5, we will introduce optimizations which lead to more efficient implementations of BCP. Even with these optimizations, BCP is still the dominant procedure in the algorithm based on its share of the total execution time.

2.3 Conflict Analysis and Learning

After the DPLL algorithm encounters a conflict, the algorithm reacts by backtracking. A conflict indicates that a satisfying solution cannot be found in a particular region of the search space. The basic DPLL algorithm does not try to analyze the conflict any further. In this case, the algorithm simply restricts itself to detecting the conflict. Then it immediately tries to visit a new part of the search space solely based on the content of the decision stack. To accomplish this task, the algorithm keeps track of which decisions in the search space have been flipped. Those decisions which are not flipped can be tried using the complementary value. Flipped decisions on the other hand are simply removed when they are on the top of the stack when backtracking.

Such backtracking is usually effective when used in randomly generated SAT instances [64]. SAT instances created by practical applications are usually not random but rather have a structure. Thus chronological backtracking is not the most effective method to take advantage of such structure. Instead there are other methods which can be employed to backtrack to higher levels in the search tree by removing multiple levels at once. Backtracking by undoing multiple levels is called *non-chronological backtracking*. In practice, this form of backtracking is more efficient because it prunes the search space faster.

An optimization that allows non-chronological backtracking is *learning*. The learning process analyzes a conflict when it occurs and stores the information about the conflict in the form of new clauses. These clauses represent redundant information since they can always be deduced from the set of clauses in the original SAT problem. These *learned* clauses do not change the satisfiability solutions of the problem. The addition of new clauses after a conflict is called *conflict-driven learning*. Thus learning enables the augmentation of the initial formula with additional implicate clauses that are deduced during the search procedure.

Learning [90, 61, 95] was first used in SAT solvers in 1996 [95]. It was, however, first used in Constraint Satisfaction Problems [78]. The addition of learned clauses restricts the search space by preventing the solver from retrying those parts of the search tree which previously led to conflicts. Because learned clauses are deduced from the initial formula they can be discarded without changing the solution set of the initial problem. In the next section we describe how learned clauses are constructed.

2.3.1 Learning and Implication Graphs

In DPLL with learning, new implicate clauses are deduced after the cause of the conflict is analyzed. Conflict analysis is based on implication graphs. An *implication graph* is a DAG that expresses the implication relationships of variable assignments. An example implication graph is shown in figure 2.3. The vertices of the implication graph represent assigned variables. The incident edges on a vertex originate from those variables that triggered the implication of the represented variable assignment. The implication graph is not maintained explicitly in memory. Instead, each implied variable points to the unit clause that caused its implication (i.e. caused this variable to assume some truth value). This clause is called the *antecedent* of this variable. Note that decision variables have no antecedents because they are not implied. In practice decision variables are given a fictitious antecedent clause. Initial and learned clauses are given indices greater than or equal to 1, thus we use clause 0 (which does not exist) as antecedent for decision variables.

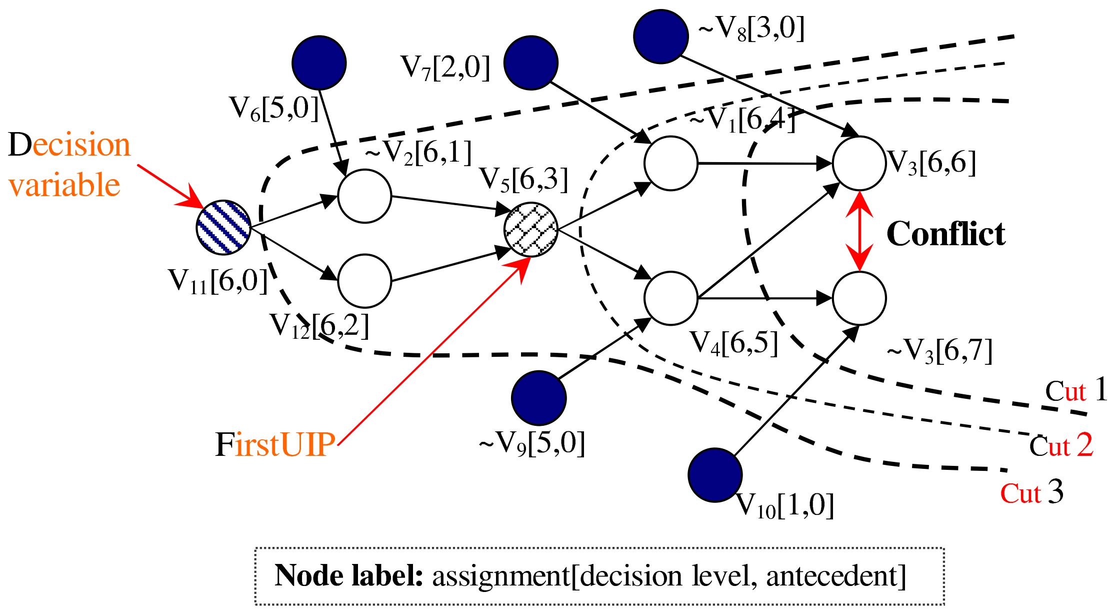


Figure 2.3: Example showing different possible cuts through an implication graph

However not all cuts generate clauses which lead to a more efficient algorithm. A cut must be selected in order to make learning effective [65] in improving the algorithm's performance. A trivial cut, such as "cut 3" in figure 2.3, would result in a clause that includes all the previous decision variables made before reaching the current conflict. Such cuts are equivalent to simple backtracking. In fact simple backtracking by marking whether decision variables are flipped is a more cost effective representation memory wise. Instead of storing a new learned clause after each backtracking a simple flag is used. Storing learned clauses in this case consumes additional memory and creates overhead during BCP. Thus some ill-chosen cuts can generated large clauses which waste memory and are not effective in pruning the search space. Carefully selected cuts would have fewer intersections and therefore will produce a smaller clause. Smaller clauses have been shown to be more effective in pruning the search space than longer ones [28].

A *learned clause* is obtained by partitioning the implication graph into two sides using a *cut*. Figure 2.3 shows three possible cuts for the shown implication graph. For a given cut one partition is called the *reason side* and contains all the decision variables. The other partition which contains the conflict is called the *conflict* side. The cut is used to generate a new *learned* clause using the vertices (representing literals) on the reason side whose edges intersect the cut. The choice of the partitioning method determines the learning scheme generated.

The purpose of the new clause is to prevent, in the future, the set of simultaneous assignments which led to the current conflict. A learned clause can also participate in the generation of new learned clauses. The new learned clause is obtained by using the complement of the variables on the reason side. In addition, a conflict clause causes the solver to perform a non-chronological back-jump. After back-jumping, the new decision level is the highest decision level among all the decision levels of the variables in the new learned clause.

Non-chronological backtracking allows the algorithm to backtrack by undoing multiple levels at a time. Also some decision levels may grow as more variable implications are added to it because of back-jumping. Sometimes the algorithm can backtrack to the first decision level. After the conflict is resolved a new variable is added to this level. In this way the first decision level can grow in size as more variables are assigned final values. Finally the algorithm terminates by finding a satisfiable solution or running into conflict at the first decision level. A conflict at this level cannot be resolved because all values there are assigned final values. Thus when a conflict occurs at this level, the algorithm concludes its investigation declaring the problem to be unsatisfiable.

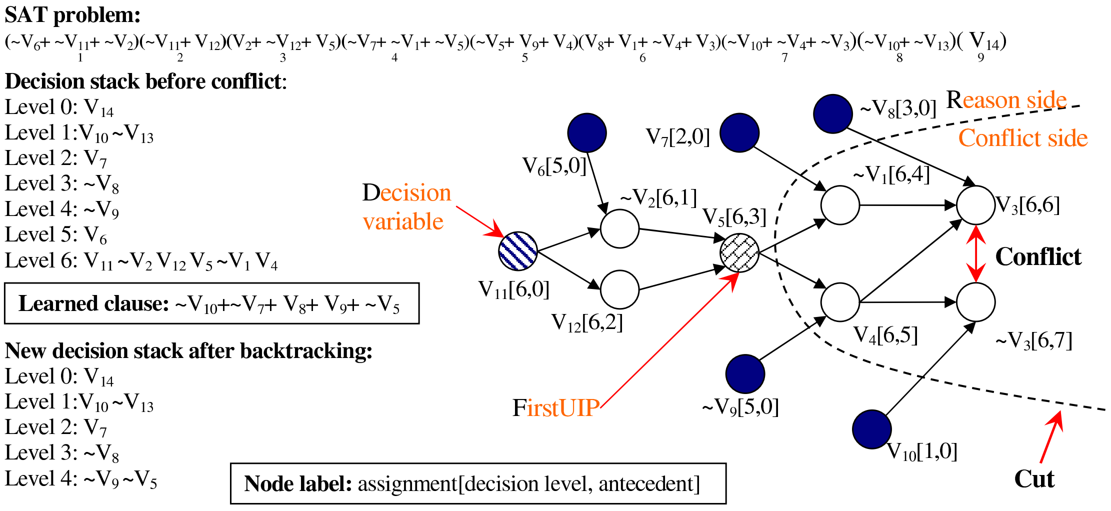
In order to construct small efficient clauses, Chaff [70] uses a method called *FirstUIP*. This method is based on finding a *dominant node* to the conflict nodes. A dominant node is a node in the implication graph such that it belongs to all paths from the vertex representing the current decision variable to the conflict vertices. The variable corresponding to the selected dominant node is the only variable added to the learned clause besides the set of decision variables. Since there might be many such nodes, the FirstUIP method uses the node closest to the conflict. In this case, the cut is made such that all implications between the dominant node and the conflict vertices are on the conflict side.

In the next section, we give a detailed example and explanation of how conflict clauses are constructed and how non-chronological back-tracking is performed.

2.3.2 Example Execution of DPLL Algorithm with Leaning

The SAT formula for this example is shown at the top of Figure 2.4. It consists of nine clauses and fourteen variables. We start with an empty decision stack and with current decision level set to 0. Since clause 9 is a unit clause, then variable V_{14} is set to *true* as it must be true for the original problem to be true. It is put in level 0 because this assignment should hold if the problem is to be satisfiable. Since there are no implications, we make a new decision level and push it on the decision stack. We choose (arbitrarily in this example) to set V_{10} to *true* and add it to level 1 as a decision variable. Like all other arbitrary decisions in this example, this decision might not be optimal but it is just used to illustrate how a SAT solver functions. After V_{10} is set to *true*, clause 8 only has one unknown $\sim V_{13}$ while the other literal V_{10} is already assigned the value *false.* The assignment of *false* to V_{10} leads to an implication and $\sim V_{13}$ is set to *true.* Because $\sim V_{13}$ is an implication it is added to current decision level (level 1). We use this same procedure until we get to level 6. In level 6, we decide to set V_{11} to *true* making V_{11} the decision variable for that level. This results in a cascading series of implications that lead to a conflict. The implication graph in Figure 2.4 shows how the implications cascade. The black nodes $(V_6, V_7, {\sim}V_8, {\sim}V_9, V_{10})$ represent previous assignment decisions, whereas the white ones represent implications at the current decision level. The conflict as shown in the graph is due to V_3 being implied to both *true* and *false* because of clauses 6 and 7 respectively. The FirstUIP node is V_5. It is a node through which all paths from the decision variable at the current level (V_{11} in the figure) to the conflict nodes must pass.

The implication graph in the figure shows how zChaff would make a cut. Other learning algorithms would construct cuts that are different from zChaff's, generating different learned clauses for this example. The subject of cut determination is an active research area among competing SAT solvers [70]. The figure also shows the conflict and reason sides defined by the zChaff cut. All decisions that have edges intersecting the cut and the implication point (V_5 shown crosshatched in the figure along the path from V_{11} to the conflict point) represent the reason for this conflict. Thus, we learn from this conflict that $V_{10}{\cdot}V_7{\cdot}{\sim}V_8{\cdot}{\sim}V_9{\cdot}V_5$ should not all be *true* simultaneously making the new learned clause ${\sim}V_{10} + {\sim}V_7 + V_8 + V_9 + {\sim}V_5$. We then backtrack to the maximal decision level of all the decision variables involved in the conflict. This level is 4 which is the decision level of $\sim V_9$. The new decision stack is also shown in Figure 2.4. Note that when using this method the new learned clause leads to an implication after backtracking involving the FirstUIP variable(V_5). In this implication the FirstUIP node V_5 is set to *false*.



SAT problem:

$\underset{1}{(\sim V_6 + \sim V_{11} + \sim V_2)}\underset{2}{(\sim V_{11} + V_{12})}\underset{3}{(V_2 + \sim V_{12} + V_5)}\underset{4}{(\sim V_7 + \sim V_1 + \sim V_5)}\underset{5}{(\sim V_5 + V_9 + V_4)}\underset{6}{(V_8 + V_1 + \sim V_4 + V_3)}\underset{7}{(\sim V_{10} + \sim V_4 + \sim V_3)}\underset{8}{(\sim V_{10} + \sim V_{13})}\underset{9}{(V_{14})}$

Decision stack before conflict:
Level 0: V_{14}
Level 1: $V_{10} \sim V_{13}$
Level 2: V_7
Level 3: $\sim V_8$
Level 4: $\sim V_9$
Level 5: V_6
Level 6: $V_{11} \sim V_2\ V_{12}\ V_5 \sim V_1\ V_4$

Learned clause: $\sim V_{10} + \sim V_7 + V_8 + V_9 + \sim V_5$

New decision stack after backtracking:
Level 0: V_{14}
Level 1: $V_{10} \sim V_{13}$
Level 2: V_7
Level 3: $\sim V_8$
Level 4: $\sim V_9 \sim V_5$

Node label: assignment[decision level, antecedent]

Figure 2.4: Example of conflict analysis with learning and non-chronological backtracking

2.4 Memory layout

Current SAT solvers are used to solve very large instances with thousands and even millions of clauses. Storing the initial set of clauses alone requires a sizable chunk of memory. It is not uncommon for the initial set of clauses to require over a few hundred Mega Bytes of memory. In addition, the number of learned clauses during execution is potentially very large and often requires more memory than the initial set of clauses. Clauses are kept in memory and are not stored on disk to avoid the slowdown of BCP due to long delays when accessing information stored on disk. Thus, using disk for storing clauses would significantly slow down the solver. In addition, not all learned clauses can be stored with the initial clause set because it requires large memory size, thereby consuming and ultimately exhausting the memory capacity of any given host.

Effectively managing the memory used to store the large set of clauses is critical to the performance of the solver. A major concern for SAT solvers is the memory layout of the data structure used to hold clauses and its effects on the efficiency of the algorithm. A well designed memory layout of the clauses is important because the clause set is frequently accessed, added to or deleted from. The most important of these operations that needs be targeted by the optimization of the memory layout is the access of the clauses. Clause access occurs primarily during BCP – by far the most costly operation in SAT solvers.

Some early solvers used linked lists and array of pointers to store the clause set. The motivation was that pointer heavy data structures are convenient to navigate, add to and remove from. However, these structures when updated frequently usually leads to fragmentation of the clause set in memory. Therefore, accessing the clauses does not exhibit locality of memory reference. As a result, implementations using pointer heavy structures suffer from frequent cache misses resulting in degraded performance.

In most newly developed solvers, a common memory layout for storing clauses is a single-dimensional array. The array structure allows for efficient access to the clauses while incurring minimal storage overhead compared to linked lists. Storing clauses in an array makes use of locality of reference which improves cache performance, thus resulting in a considerable speed-up of the solving process. In section 2.5 we show how access to the clauses is further improved by reducing the set of clauses visited during BCP. Maintaining the array structure needs additional code for garbage collection and organization. This code is only invoked intermittently which makes for very low execution overhead.

2.4.1 Space Management

The host machine of a solver need sufficient memory to store a large set of clauses. If the memory becomes scarce, a SAT solver continues to make progress albeit very slowly. The slowdown is caused by the decrease in the solver's ability to store newly learned clauses which are essential to pruning the search space. In this case more memory space needs to be made available to store newly learned clauses which are more relevant to the current part of search space being investigated by the solver. Modern solvers use heuristic to delete some of the learned clauses. Deleting some of the learned clauses periodically alleviates memory use and allows the addition of new learned clauses which are currently more relevant.

In general, since all learned clauses represent redundant information, the algorithm can discard them without affecting correctness. Deleting learned clauses, however, should only be performed to alleviate memory usage because it may hinder the solver's progress. Still there are some learned clauses which may not be deleted because they may be used by the learning process. These are clauses which are antecedents of variables currently in the decision stack. These clauses cannot be removed because they are used in the conflict analysis and learning for subsequent conflicts. When backtracking occurs, those variables which are unassigned are no longer bound to their antecedent clauses. These clauses do not serve as antecedents to any variables and can be safely deleted.

When a solver initially has little memory available, what little memory it can use may be consumed by antecedent clauses. This leaves little additional memory to store learned clauses. This virtually eliminates the benefits of learning. The result is that the solver's progress is hindered. In the next section we discuss Chaff specific heuristics [70] to select which learned clauses are deleted depending on their size and other properties.

2.5 Chaff Implementation

As mentioned earlier, we chose zChaff as a basis for our research because it is the most powerful sequential solver available. zChaff is an implementation of Chaff by *L. Zhang* from Princeton. There is another implementation called *mChaff* [70, 67] which was independently developed by *M. Moskewicz*. Both versions implement the same optimizations which Chaff makes but differ in the implementation details. Both versions differ in the data structures and some heuristics used to delete clauses. Chaff is a modern solver which uses all the optimizations mentioned earlier. In addition, Chaff introduces two optimizations to the basic stack-based algorithm: a more efficient method for BCP and a new heuristic for choosing decision variables. These optimization are also being adopted by other solvers [47].

2.5.1 Watched literals

As mentioned earlier, BCP accounts for a large portion of the execution time of the DPLL algorithm. BCP involves the inspection of the clause *database* in search of unit clauses, after each variable assignment. We use the term *database* in the rest of this thesis to refer to the set of clauses used by the solver. Actually, we can think of the solver as performing only a very specific *query* but very often. The query is executed after every decision or implication. The query matches all clauses which contain a specific literal. However since efficiency is of utmost importance, zChaff uses a very efficient BCP procedure to boost the overall performance. This section is dedicated to describing how zChaff implements this procedure.

An intuitive but inefficient way to check for unit clauses is to check all clauses that have one of their literals set to *false* by the last variable assignment. Thus a clause composed of n literals will be checked *(n-1)* times before it becomes a unit clause. However, a clause need only be visited when there are two unknown literals left because that is when it is likely to become a unit clause and generate an implication. Chaff uses this observation to implement the *two-literal watching scheme*. In this scheme two literals are selected for each clause and are called the *watched literals*.

In the course of the algorithm a clause can become a unit clause because of different literals, therefore the watched literals cannot be determined in advance and are not always the same. In Chaff this procedure is implemented by initially marking two literals from each clause – the first and the last one. When one of the watched literal becomes *false* because its variable has been assigned a value (i.e. *true* or *false*), then that literal is unmarked . If another unmarked literal with unknown value from the same clause exists then it is marked. In this case one of the watched literals has been modified. If no such literal exists then the clause is unit and the other marked literal is implied to be *true*. Thus after each variable assignment not all clauses containing the *false* literal are visited. Instead only those clauses where this literal is watched need to be included in the BCP procedure. This technique reduces the number of clauses visited after a variable assignment. In addition, it reduces the number of times a clause is visited before it becomes a unit clause. The overall result is a more efficient BCP procedure.

Figure 2.5 shows the memory layout of the zChaff database. The figure shows a large array that is allocated as a memory pool for storing clauses within zChaff. This layout is designed for fast access and easy update of the set of all clauses both initial and learned. The set of initial clauses is stored at the beginning of the array. The figure also shows how zChaff uses an index over the set of clauses based on the variables. This index enables efficient inspection of clauses where a particular variable (or its complement) is used. Also notice that each variable corresponds to two indices: one for positive literal and the other for its complement. Each index points to a subset of clauses where the literal occurs.

The clause lists of each index are built by adding only those clauses where the corresponding literal occurs **and** is being marked as watched in its containing clause. When a variable is assigned a value it only considers those clauses pointed by the index of that literal which evaluates to *false*. All clauses pointed to by the *true* literal component are now satisfied and need not be considered at this point. Since each literal index contains only those clauses that are being watched not all clauses are inspected during BCP. In fact only a small portion of clauses are visited after a variable is assigned a value. For example even those clauses which contain the current literal may not be visited if that literal is not being watched. By inspecting a small subset of clauses, the two watched literal scheme implements a very efficient BCP process.

When a clause is inspected during BCP, then one of the watched literals has just been assigned a *false* value. There are three possible cases.

- In the first case, the other watched literal is still unknown and all other literals are *false*. This leads the remaining unknown literal being implied as *true*.

- In the second case, all other literals are also *false*. This leads to a contradiction since the entire clause is unsatisfied. The other watched literal is also *false* because of previous implications. This is the first time BCP visits this clause after the last decision was made.

- Finally, the other watched literal is unknown and at least one non-watched literal is still unknown. In this case, one of the non-watched literals is selected and marked as watched. This literal replaces the watched literal which led BCP to visit this clause. Thus only two literals are marked as watched when BCP is done inspecting this clause. This clause will not be visited during BCP until one of the watched literals is assigned the *false* value.

During backtracking, the watched literals need not be modified, since the last pair of watched literals represent the variables in the clause that were last assigned values. When a backtrack occurs, those variables and their respective literals will be unassigned before any other variables in the same clause. Since they are now unassigned, they can be immediately used as a pair of watched literals.

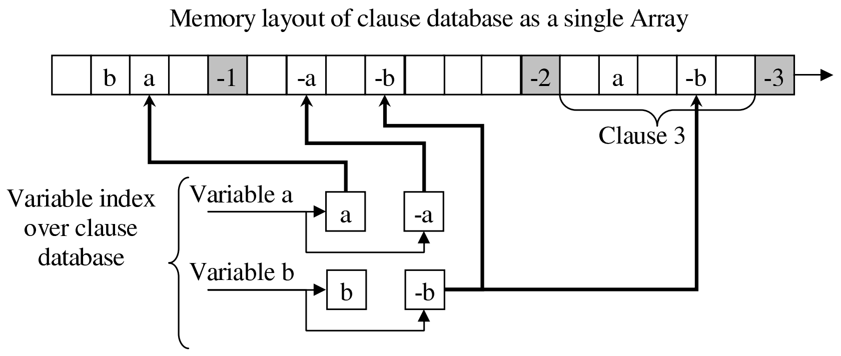

Memory layout of clause database as a single Array
b
a
-1
-a
-b
-2
a
-b
-3
Clause 3
Variable index over clause database
Variable a
a
-a
Variable b
b
-b

Figure 2.5: Memory layout of clause database and variable indexing strucuture in zChaff

2.5.2 Variable State Independent Decaying

An important heuristic in the DPLL algorithm is selecting decision variables and the values they are assigned. There are many different heuristics used by various solvers to make this selection. The general goal of these heuristics is to cause as many clauses as possible to be satisfied. Most heuristics used by solvers are based on the initial set of clauses and are mostly static. In Chaff, a more dynamic method is used.

For making decisions Chaff uses Variable State Independent Decaying (VSIDS). In this heuristic each literal (V or $\sim V$) is assigned a counter that initializes to zero. When a clause is added to the database all counters for literals occurring in the clause are incremented. The literal with the highest count is chosen for assignment. Periodically all counts are divided by a constant so that more recent clauses have more influence on the next choice than older clauses. This method was found to incur low overhead compared to other heuristics.

2.6 Summary

In this chapter, we have presented the DPLL algorithm that is the basis for most complete sequential solvers. Many additional optimizations can enhance the performance of the basic algorithm. The most effective optimizations are conflict-driven non-chronological backtracking, more efficient BCP and clause deletion. zChaff introduced a new implementation of BCP and a heuristic for making decisions which have propelled zChaff to being one of the most powerful solvers. In the next chapter, we present a parallel version of this algorithm.

Chapter 3

Parallel Satisfiability Solver

In this chapter we present the techniques we used to parallelize the sequential SAT solver. The parallel solver is based on one of the most powerful sequential solvers available today - zChaff as determined by the SAT competition [89]. We start by stating the parameters used in defining a successful parallelization. Then, we present a detailed discussion of the parallel algorithm. Next we introduce additional techniques such as clause sharing and clause reduction that improved the overall performance of the solver. In section 3.4, we present a parallel version of the solver capable of enumerating all solutions to a given SAT problem.

3.1 Parallelization Strategy

There are many alternatives to consider when parallelizing the DPLL algorithm. Our goal is to enable the most powerful parallel solver that is capable of exploiting all resources in a computational grid environment. In order to accomplish this task there are two characteristics the parallel solver must have. First, it has to make use of the extensive research in the field of sequential solvers. Thus in implementing our solver we chose to use and extend all of the optimizations used in modern solvers whenever possible. Second the solver has to be capable of adapting to the dynamic computational environment. Some of the most important characteristics crucial to the solver's performance in such an environment are memory availability for the individual resources and connectivity between resources.

There are three possible approaches to parallelizing the zChaff SAT solver. The first approach is based on one main solver that is extended by helpers. The role of each helper is to get a smaller part of the problem and communicate to the main solver any potentially helpful results. If any of the helpers stops working voluntarily or because of a failure, the main solver would not be affected. In this case helpers play a purely advisory role since they are not responsible for providing any definite answer about any part of the problem. Such a deployment does not reduce the workload for the main solver because it is still responsible for entire SAT instance. In fact, the parallel solver can be viewed as a combination of many incomplete solvers (i.e. the helpers) and one complete solver (i.e. the main solver).

The second possible deployment is using a distributed clause database. Similar to the previous solution there is only one solver deployed. The premise of this strategy is that distributing the clause set helps speeding up the time consuming BCP process. In order for this premise to be realized, the speedup realized by providing more memory to the solver has to offset communication and synchronization overhead. In a grid environment with large number of resources and highly variable computational and network performance, the overhead is potentially crippling.

The third strategy which we advocate and use in this thesis has many advantages over both previous options. In this method, the algorithm is parallelized by using multiple solvers simultaneously. Each solver is assigned its own subproblem and is entirely responsible for conclusively answering whether that part of the problem is satisfiable or not. Unlike both other options where one resource carries the entire burden, this strategy means that all resources are equally responsible for making progress in investigating the problem. Also this option reduces the search space for each of the solvers involved. This solution is also more suitable for the computational grid since the individual solvers are not tightly coupled together. This feature allows the resulting implementation to be more adaptable to dynamic performance variations in computational power, available memory and network characteristics.

3.2 Algorithm Parallelization

Parallelizing the algorithm allows multiple resources or clients to cooperate in solving the same problem. In the next section we present in detail how parallelization of the algorithm is implemented in GridSAT and some optimizations that can used to further improve the individual solvers.

3.2.1 Search Space Splitting

Search space splitting can be applied at any point during the execution of a solver. In general, this process divides a given set of possible solutions into multiple disjoint subsets. The outcome of splitting are two or more new SAT problems. The SAT problems can be solved independently. However, cooperation between solvers is beneficial as we will show later in section 3.3.

Once a problem is split, the same process can be repeated recursively. Thus the parallel solver can use as many resources as are available. Each new subproblem is defined using a set of clauses and a decision stack. Initially we can assume that the decision stacks are identical. The set of clauses for each problem contain the original set of initial clauses, the new set of clauses learned by the algorithm in addition to space-splitting clauses. Learned clauses are included because splitting can occur after the solver has made some progress which allowed it to generate new clauses.

Each of the new subspaces is defined using the space-splitting clauses. These clauses are added in conjunctive form to the entire SAT problem. In case the problem is in CNF these clauses can be treated by the solver exactly like the initial set of clauses. Since deleting these clauses changes the scope of the search space, these clauses are never deleted just like the initial set of learned clauses.

For example, if we wish the search space to be divided to two subspaces based on whether variables V_1 and V_2 are equal or different. The first search space is defined by the variables being equal while the second search space includes the rest of the search space (i.e. $V_1 \neq V_2$). Thus, for the first subspace we add the two clauses $(V_1 + V_2)(\sim V_1 + \sim V_2)$. The second problem is defined by adding $(V_1 + \sim V_2)(\sim V_1 + V_2)$. In this example the search space is divided in half. It is possible to divide the problem into different sizes. In some cases the search space is divided in a manner where certain variables are assigned specific values. In this case instead of adding single literal clauses, the new assigned variables are simply added immediately to the decision stack at the first decision level.

The parallel algorithm adopted by GridSAT allows a given SAT instance to be split into two subproblems. The splitting is conducted around a particular variable called the *pivot variable*. Each of the two subproblems assumes one of two possible values of the pivot variable. In this case, one of two new single literal clauses V_i or $\sim V_i$ (where i is the variable index) can be added to each of the new subproblem. A much simpler solution to accomplish search space splitting is shown in section 3.2.2.

Consider again the same example from figure 2.2. If the problem is divided using variable V_1 as pivot variable, then the entire problem search space is divided in half as shown in figure 3.1.

The pivot variable cannot be selected from level one since those variables have been deduced to have specific value. The pivot variable should be selected to minimize perturbation to the solver that is initiating the splitting procedure. For

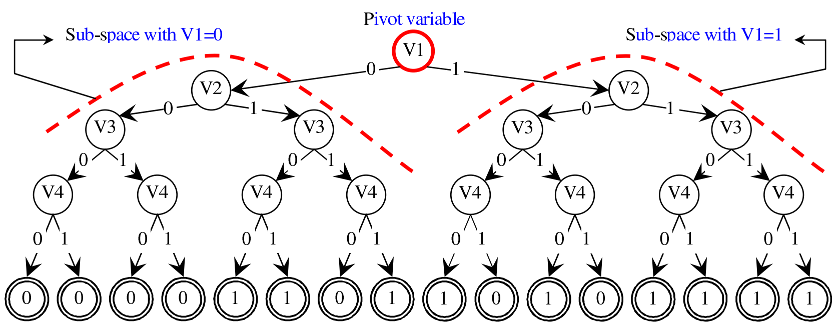


Figure 3.1: Illustration of search space split using a pivot variable

this reason we select the decision variable on the second decision level. As we will see in the next section this leads to an easy definition of the decision stacks of the two new subproblems.

3.2.2 Decision Stack Construction

During the splitting process, the desired goal is to start a new solver while keeping the other solver running on the same resource with as little delay as possible. Selecting the first variable in the second decision stack as the pivot variable, the splitting process can be accomplished by simple manipulation of the decision stack. As a result the solver initiating the splitting is allowed to continue without undoing any of the progress it has achieved so far.

Figure 3.2 shows an example of the split process which starts with a SAT problem being solved by client A. The new created subproblem is spawned and

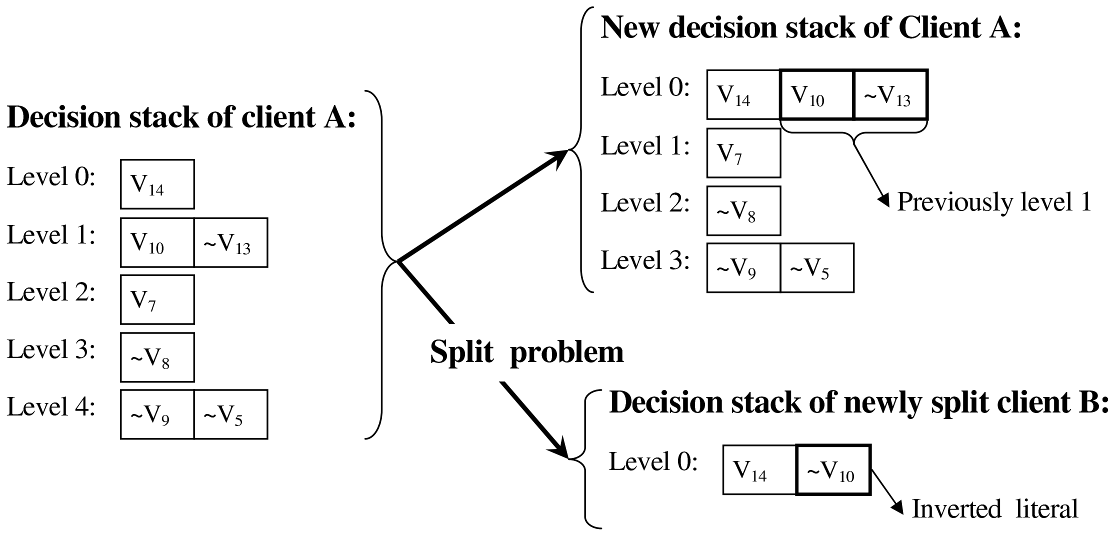

Decision stack of client A:
Level 0: V_{14}
Level 1: V_{10} $\sim V_{13}$
Level 2: V_7
Level 3: $\sim V_8$
Level 4: $\sim V_9$ $\sim V_5$
Split problem
New decision stack of Client A:
Level 0: V_{14} V_{10} $\sim V_{13}$
Level 1: V_7
Level 2: $\sim V_8$
Level 3: $\sim V_9$ $\sim V_5$
Previously level 1
Decision stack of newly split client B:
Level 0: V_{14} $\sim V_{10}$
Inverted literal

Figure 3.2: Example of stack transformation when a problem is split into two clients.

assigned to client B. The old problem is modified by making all variables on the second decision level of the assignment stack part of the first decision level. All the variables on this level have been deduced from the assigned value of the pivot variable. Thus if the pivot value has assumed that same final value because of splitting, the rest of the variables on the decision stack will also assume their respective final values. Since all these variables are assigned final values, they are all stored at the first decision level. In figure 3.2, V_{10} is the pivot variable. For the old client V_{10} is assumed true, thus V_{13} is now false according the second decision level. Both V_{10} and $\sim V_{13}$ become part of the first decision level in the context of client A. All other levels of the decision stack are shifted up one level but are not modified. The solver in client A can continue by making a decision at level 4.

The new problem generated consists of a set of variable assignments and a set of clauses. The decision stack for this problem is based on the initial decision stack of client A and the pivot variable. The stack for the new client has only one decision level. The variable assignments on this level include all assignments from the first decision level and the complement of the first assignment in the second decision level, thus ensuring the splitting of the search space. As shown in figure 3.2, client B assumes V_{10} to be false. Thus $\sim V_{10}$ is added to the first decision level of client B. The solver in client B starts making new decisions at the second decision level.

3.2.3 Clause Reduction

After splitting based on a pivot variable, each process maintains its own separate clause database. Since both processes have assumed new final values for one or more variables, some clauses become inconsequential. These are clauses whose final values are now fixed. In fact, if these clauses are kept, the algorithm will never refer to them again. Thus they are simply wasting valuable memory. These clauses are different for both new sub-problems and include both initial clauses as well as learned clauses. Removing these clauses alleviates memory usage by learned clauses and causes the initial set of clauses to be reduced. Reducing memory usage allows for more more memory to store learned clauses. As a result the solver's performance is improved because more learned clauses can prune a larger portion of the search space.

The inconsequential clauses evaluate to *true* because they contain literals which are also *true* according to the assignments on level 0 of the local decision stack. In the example in figure 2.4 client A can remove clauses 8 and 9 because their respective literals $\sim V_{13}$ and V_{14} are *true*. Client B can remove clauses 7, 9 and also the newly learned clause because their respective literals $\sim V_{10}$, V_{14} and $\sim V_{10}$ are *true*. For the new client B, the set of inconsequential clauses can be determined before the entire clause database is sent. Therefore, inconsequential clauses are not sent to the new client in order to reduce communication overhead.

This memory saving technique can also be applied to a sequential solver each time the first decision level is augmented with new assignments. Because scarcity of memory is often the limiting factor, we also implemented this procedure in the sequential version of zChaff that we use for comparison.

3.2.4 Sending Clause Database

Part of the splitting process of the search space with another client involves sending a set of learned clauses. The set of learned clauses could be very large and therefore very expansive to communicate. Sharing these clauses, however, improves the solver by providing valuable pre-computed information that the solver can then immediately use to prune the search space. The desired strategy is to send the smallest number of clauses which will be most useful to the solver. The solution chosen is evaluated based on the speedup the solver gains. Since the speedup is not usually guaranteed it is desirable to choose a solution with low overhead.

There are many possible heuristics that can be used to select which sets of clauses are shared with the new client. One such approach is to only send a random fraction of the entire set of clauses. Another possibility is to rank the clauses based on their age and only send the newest portion of the clause database.

The heuristic we use is based on the observation that the smallest clauses are the most useful in pruning the search space. Thus only a fraction of clauses with the smaller sizes are shared. This solution uses only one parameter which is the fraction of clauses to be shared, we call it the *splitting fraction*. The implementation finds the corresponding maximum clause size for this parameter. This is accomplished by sorting all clauses sizes in the database in increasing order. Once the maximum clause size is determined only those clauses smaller than this clause size are shared.

3.2.5 Ping-pong Effect

A possible risk in parallelizing a SAT solver comes from the possibility of excess overhead introduced by parallel execution. In particular, because the duration of execution time that will be spent to solve a subproblem cannot be predicted easily beforehand, it is possible for subproblems to be investigated in such a short amount of time that the overhead associated with spawning them cannot be amortized. As a result a solver spends more time communicating the necessary subproblem descriptions, thinning the database, and collecting the results than it does actually investigating assignment values. Even though the solver is advancing, the execution time will be longer than if it were executed sequentially. This problem is occasionally referred to as the “ping-pong” effect [58]. This risk is mitigated at the implementation level as shown later in chapter 4.

3.3 Clause Sharing

Although all subproblems are assigned disjoint parts of the search space they are still working on very similar and related problems. The different subproblems obviously may still share some of the initial clauses. In addition, the solvers might be making speculative decisions about common variables. Thus it is probable that they will learn information which can be shared.

In a parallel solver each process is given a unique top level in the decision stack after problem splitting occurs. However, this level usually contains a small number of variables compared to the total number of variables especially at the onset when splitting is most likely to be used. Therefore, a large number of the variables still under investigation are shared amongst subproblems. Such variables are called *active variables.*

The different solvers working on the various subproblems can learn new clauses about a common set of active variables. These solvers will learn different clauses because they will be making different decisions as a result of the status of their respective decision stacks, existing clauses and heuristics. Even though the learned clauses are generated by investigating parts of the search space defined partially by the initial decision stack, the clauses may not explicitly depend on the decision stack. An explicit dependence means that some of variables in the initial decision stack (or their complements) are part of these clauses. Because of the large number of variables, newly learned clauses may not contain variables from the initial decision stack.

Therefore, after the initial problem is split into many new subproblems, each of these subproblems will produce new learned clauses using a shared set of variables. Since all the subproblems are still investigating the assignment space for these variables, knowledge discovered by one process can be used in the context of other subproblems. According to the learning process in section 2.3, each of the new learned clauses is a direct logical deduction from the initial set of clauses. Therefore, these learned clauses are logically independent from the decision stack. Actually, any learned clause can be used as-it-is within any other solver at any time. Thus when these learned clauses produced by one client are shared with other clients they help prune parts of their search space which they have not yet investigated. In addition, by obtaining these clauses a receiving client saves the time and resources needed to discover this information on its own. The overall effect is improved solver performance.

Allowing clause sharing, however, limits the kind of simplifications that can be made. For example, variables (and their complements) which have known truth assignments (i.e. in the first decision level) can be removed since they will not influence future decisions made by the solver. Removing such variables can be accomplished by deleting the occurrence of all literals with known values from all clauses. This deletion results in shorter clauses and more efficient use of the memory. However, the resulting clauses are not anymore logically independent of decision stack. Thus sharing clauses modified in this manner leads to incorrect results because variables of known values in one process might still be unknown in another process. Thus in order for a clause to be still valid when shared with another process it must contain complete variable information. Therefore simplifications such as removing known variables are not possible when clauses are shared because they make learned clauses only valid in the context of the current solver.

When new learned clauses are received from other clients, they are merged with the local clause database. The merging process has to be accomplished in a way that is logically consistent. Adding a clause directly to the database may cause logical inconsistencies if that clause contradicts some of the deductions previously made by the algorithm. For example, a variable may have been deduced to have a true value. However, if the learned clause was known to the algorithm at the time of that deduction the algorithm may have run into a conflict. Thus adding received clauses from other clients may include undoing some of the previous decisions or adding new ones. We have investigated three merging strategies. In the next three sections we present the three methods that GridSAT uses for sharing learned clauses.

3.3.1 The Lazy Method

The *lazy method* is the simplest method for making use of learned clauses because it limits the merging of newly obtained clauses into the clause database to certain phases of the DPLL algorithm. These phases occur after the algorithm has backtracked to the first decision level. In this case, merging the new clause does not involve any stack manipulation because the stack contains one level and no speculative decisions. The only variables to take into consideration are in the first level of the stack. The truth values of these variables are never altered by subsequent decisions thus no speculative decision need to be undone.

Under the conditions outlined previously, merging a received clause is straight forward. Figure 3.3 shows how the DPLL algorithm is modified to enable this method of merging clauses. The DPLL algorithm is augmented by adding a simple test before each new decision to check if the algorithm has backtracked to the first decision level. If the test fails, the algorithm continues as before. If

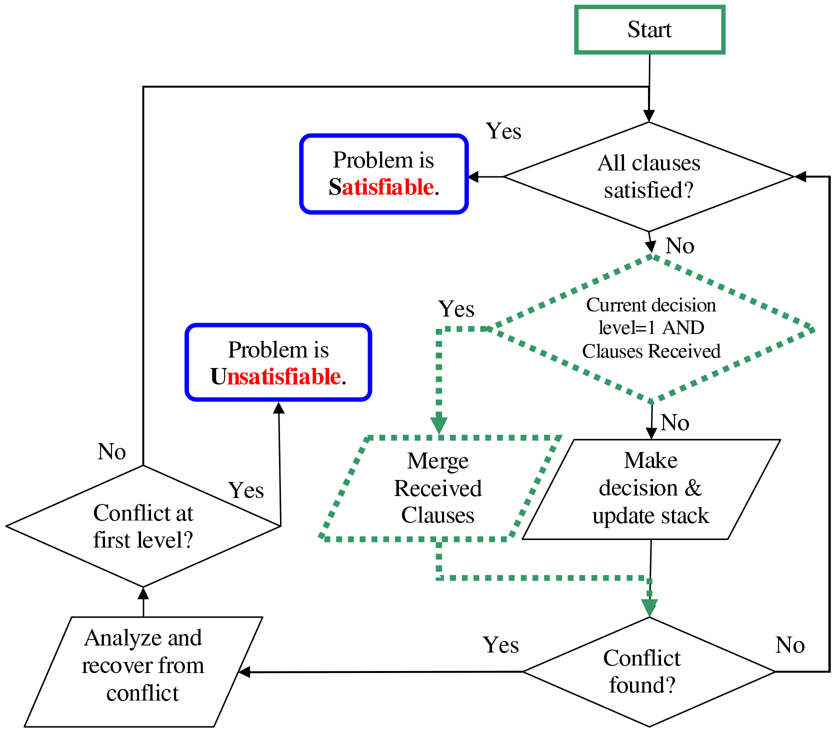

Start
All clauses satisfied?
Yes
Problem is Satisfiable.
No
Current decision level=1 AND Clauses Received
Yes
No
Merge Received Clauses
Make decision & update stack
Conflict found?
Yes
No
Analyze and recover from conflict
Conflict at first level?
Yes
No
Problem is Unsatisfiable.

Figure 3.3: DPLL algorithm modified to enable lazy clause merging

the test succeeds however, the merging process is activated. During the merging process each of the clauses received are processed one at a time. The literals of the received clause are examined for their truth values which can be either *true*, *false* or *unknown*. For a given clause there are four possibilities:

- If the clause contains at least one *true* literal, then the entire clause is *true*. Since the decision stack contains no speculative decisions, the variable corresponding to the *true* literal could have only come from the first decision level. Since this variable will always be *true*, the clause will always be satisfied. Therefore the clause is of no value to the solver since it does not help restrict the search space and is discarded. In the rest of the cases we assume that no literal is *true*.

- If the clause has only one *unknown* literal and the remaining literals are *false*, then an implication is generated. The newly implied variable assignment, is therefore predicated only on variables on the first decision level. Thus the implied variable is added to the first level of the decision stack. The clause under consideration is marked as the antecedent for the newly implied variable.

- If the clause has more than one *unknown* literal then the clause can be used to restrict the search space. In this case the clause is added to the set of learned clauses and the decision stack is not altered.

- If the clause has all literals set to *false* then this clause is not satisfied by the existing variable assignments and a conflict exists. Since the decision stack contains no speculative decisions, then all the variables in the new clause must be in the first decision level. Therefore, we have a conflict because of variable assignments which should be *correct* if the subproblem were satisfiable. Thus the subproblem is unsatisfiable.

The clauses are processed in batches where no BCP is performed until all clauses in the same batch are inspected. During the batch processing, some clauses may be added to the database while new implications are saved to a temporary queue. If a clause in the batch causes a conflict then the solver terminates immediately. However, if there is no conflict after all new clauses are processed, the solver continues by retrieving the queued implications one at a time, adding them to the first decision level and performing BCP as described earlier. Like other implications, the ones caused by merging the received clauses can also lead to a cascade of implications.

3.3.2 The Immediate Method

The lazy method is simple to implement and has low overhead, but it is ineffective in many cases. For many hard problems it may take a long time before the solver would make enough progress to cause it to backtrack to the first decision level. In this case, the solver is not able to use the clauses received from other processes. Therefore, all the shared clauses accumulated by the local solver are wasting valuable memory space since they are never used. Thus sharing clauses does not have the desired effect of helping to prune the search space of the local solver. Instead performance is degraded because of wasted memory space. This problem could be solved by allowing immediate integration of received clauses into the solver's clause database. Thus this method is called the *immediate method*. The modified DPLL algorithm for this method is shown in figure 3.4. It is similar to the lazy method in figure 3.3, except that the decision level test is removed.

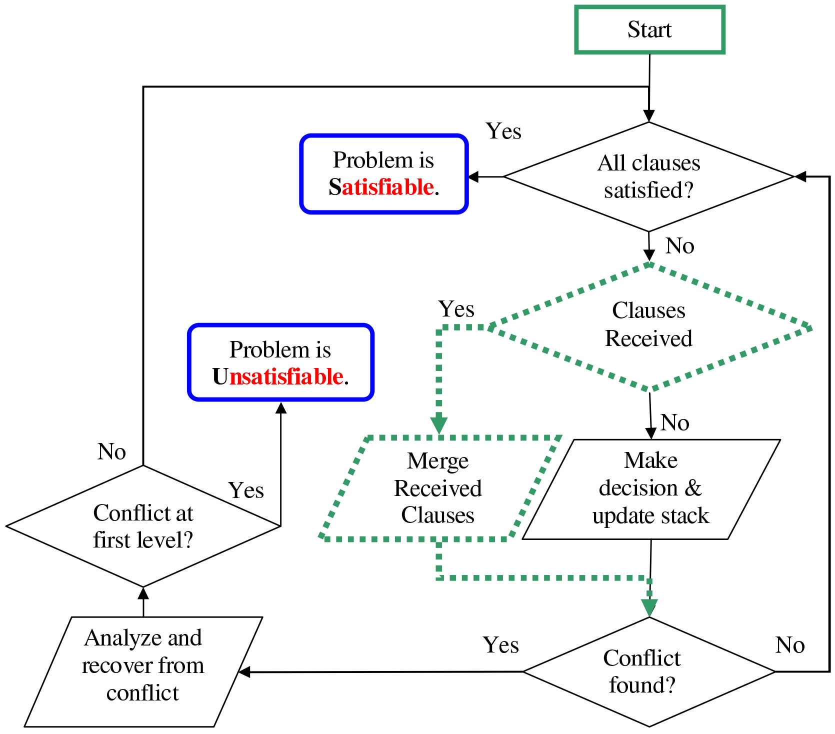


Figure 3.4: DPLL algorithm modified to enable immediate clause merging

The implementation of the immediate method is more complex compared to the lazy method described earlier because it involves complex manipulation of the decision stack. The complexity originates from ensuring logical consistency while modifying on-the-fly the decision stack which may include multiple levels of speculative decisions. The smallest granularity at which the immediate method is activated is before each new decision is made. This happens frequently in the span of the execution of the DPLL algorithm. The merging procedure is only activated if new clauses have been received and are waiting to be merged.

When a new clause is about to be merged it might not be logically consistent with implications and decisions in the decision stack. For example, the new clause might cause a new implication in an earlier level of the decision stack. It may also conflict with some implication at any level in the decision stack. This kind of decision stack manipulation is complex but it has a simple underlying principle. For each clause, the effect (i.e. implication, conflict or none) of the new clause is evaluated. Then the decision stack is rolled back and the sequence of implications or conflicts is resolved as if the new clause has been known to the solver all along.

The algorithm for merging clauses starts by inspecting the newly obtained clauses. The algorithm determines how many literals in the clause have values *true*, *false* or *unknown*. Also the algorithm determines for clauses with a single literal being *true*, the decision level *true_lit_dl* of such a literal. For the given clause it determines the maximum decision level (*false_lit_max_dl*) amongst the decision levels of the literals set to *false*. After determining these value there are only five possible outcomes:

- If the clause is satisfied because of a variable assignment at the first decision level, then this clause is useless for the local solver and is discarded. This case is similar to the first case in the old merging algorithm.

- If the clause has only one *unknown* literal and no *true* literals, then the clause results in an implication. Actually if the clause was available when the solver was still generating implications for *false_lit_max_dl* decision level, then this clause would have become a unit clause and would have generated an implication. Because generating implications as early as possible is very important for directing the search, we allow the solver to backtrack in order to make use of this implication. In this case, the solver backtracks to decision level *false_lit_max_dl* and the clause is inserted to the clause database. After the solver backtracks to *false_lit_max_dl* decision level, the same previous speculative decision at this level is put in temporary queue.

- If the clause has only one *true* literal and no *unknown* literals, then if *false_lit_max_dl* is smaller than *true_lit_dl* then this is indeed an implication. This restriction is necessary because there might be cases where the clause has only one *true* variable but it does not represent an implication. In such cases the *true* variable was set at a level while some of the remaining literals were *unknown* but are now set to *false*. The solver proceeds by backtracking to *false_lit_max_dl* and queuing an implication in the same way as the previous case.

- If the clause has all its literals set to *false*, then the clause has resulted in a conflict. In fact if this clause was available when decision level *false_lit_max_dl* was still being populated by implications then this clause would have caused a conflict at this level. This conflict would have helped direct the search, if detected. Thus the solver backtracks to make use of this conflict. However, if the conflict is at the first decision level then this situation is the same as the fourth case in the previous merging algorithm mentioned above. Therefore the sub-problem is unsatisfiable. If the conflict is at a higher level then the solver backtracks to *false_lit_max_dl*. Also previous decision at this level is saved in a temporary queue in the same way as the previous two cases.

- If none of the above cases apply then the clause is added immediately to the clause database without altering the decision stack.

If many clauses are received during the time interval between two consecutive decisions, the immediate method needs to merge them all at once. During the merging of each new clause, the decision stack is modified and a backtrack is performed in three of the five cases presented above. In addition, every backtrack results in reduction of the decision stack depth unless the top level is reached. When the stack depth is reduced, the implication queue is cleared before any new implications are added. Also the decision level from which the solver will start (i.e. *false_lit_max_dl*) is also cleared. The solver then proceeds to reconstruct the resulting sequence of implications while taking the new clauses into consideration. When the solver backtracks to the first level in the decision stack, the immediate method becomes equivalent to the lazy method since no backtracking can be done beyond this level.

The effect of backtracking to a higher level in the decision stack helps the solver investigate a more relevant part of the search space due to the newly found implication or conflict. The merging of shared clauses from other solvers restricts the search space and prevents the solver from wastefully revisiting some parts of the search space. Merging new clauses has an effect similar to randomization.

Randomization is a process where the decision stack is cleared after a timeout period and then starts at another random location in the search space. Thus if the solver does not makes progress within a given period of time it should be directed to investigate possible solutions elsewhere in the search space. The solver keeps all learned clauses but the decision stack is cleared except for the first decision level. The hope is that the restart will lead to a better location in the search space which will help solve the problem faster. Randomization is used by most solvers and has been shown to improve solver performance. Unlike randomization, merging new clauses helps choose more relevant parts of the search space based on new knowledge and not based on random chance.

3.3.3 The Periodic Method

The lazy and immediate methods for sharing clauses have two types of overhead. The first, is communication overhead which is the same in both methods. The second is solver interruption overhead which is incurred because the solver is momentarily stopped to allow clause merging to happen in a consistent manner. This type of overhead is more significant for the immediate method since it interrupts the solver frequently and at a very fine grain level in the main DPLL algorithm loop.

The communication overhead can exacerbate the interruption overhead. For example, when a large number of clients are sharing even a small number of clauses the total communication overhead becomes significant. Shared clauses could be streaming into each solver occasionally at high rates, especially if the number of processes used is high. Therefore merging the clauses immediately causes frequent preemption of the solver. When the solver is preempted it stops until the received clauses are merged. In order to decrease the rate of solver preemption, there is a third merging method that can be used. This is called the *periodic method* where the solver is parameterized to allow clause merging only after a fixed number of iterations or a fixed time interval. This results in the clauses being merged in batches in a similar fashion to the lazy method.

3.3.4 Clause Duplication

There is a chance that some of the newly merged clauses which are added to the clause database can be duplicates of other previously existing clauses. Only clauses which do not result in implications or conflicts can be duplicates. Duplicate clauses will waste valuable memory space. However, checking each new clause received by a solver to ensure that it is not a duplicate before adding it to the database is computationally expensive. It requires scanning the entire database and comparing the new clause with every clause in the database. However, since GridSAT broadcasts clauses immediately after they are learned, then all solvers become aware of the new clause quickly. Once a solver has a copy of the clause in its database it will not re-learn it. Therefore, there it is unlikely that duplicate clauses will be an overwhelming problem.

3.4 Solution Enumeration

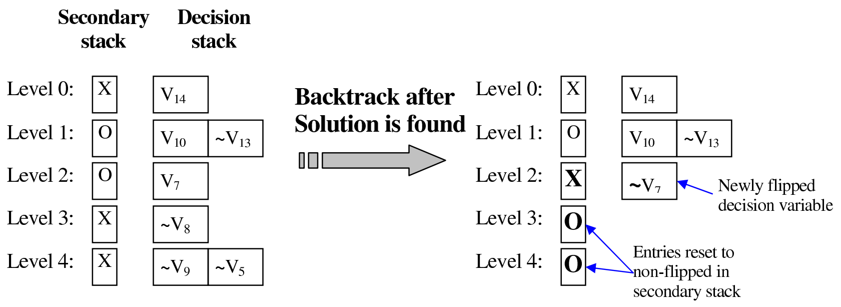


Figure 3.5: Solution Enumeration
: Example decision and secondary stack modifications after a solution is encountered. In the secondary stack X and O stand for flipped and not flipped respectively.

There are many related problems to satisfiability which are also computationally intensive, such examples include $\#SAT$ and *solution enumeration*. The $\#SAT$ problem only asks to determine the number of solutions for a given SAT problem instance. Solution enumeration problems, however, require the listing of all solutions to a SAT problem and not just the number of solutions or whether the problem is satisfiable or not. The solution enumeration problem is important because in many cases it is desirable to find all solutions to a problem or at least a representative subset of the solution set. In [56], the author presents a motivation for solution enumeration and how it can be used to improve software testing procedure. Similarly, solution enumeration could generate multiple solutions to a scheduling problem [11]. These solutions would present alternative solutions to choose from instead of being restricted to a single one. In another example, a circuit designer with access to multiple solutions can select the solution that best suits his needs. Moreover, in cases where satisfiable solutions represent design errors multiple solutions provide more information about the sources of error and may lead to quicker determination of the source of error. Solutions to the $\#SAT$ and solution enumeration problems can be derived from solutions to the original satisfiability problem.

The basic DPLL algorithm terminates after the first solution is determined. However, a satisfiability problem may have more than one solution. There are multiple ways to alter a DPLL-based sequential solver in-order to enable solution enumeration. For example, a simple approach would be to augment the initial set of clauses with a clause for every solution encountered. These clauses are never deleted just like the initial set of clauses defining the SAT instance. The addition of each solution clause would prevent the solver from generating the same solution in later steps. Such clauses are usually long because satisfiable instances often include most of the variables. A major drawback of such a solution is the need to use more memory to store all these clauses. As the number of solutions is usually high, the memory needed to store the clauses produced by solutions becomes very large. This makes the solver less efficient as less memory is available for clauses obtained through learning.

GridSAT with enumeration uses a different approach that incurs little memory overhead as the number of solutions found increases. This method could easily be integrated to other DPLL-based solvers and is not specific to zChaff. In the next section, we describe this method as it applies to a sequential solver. In section 3.4.2 we present how this method can be incorporated into the parallel solver.

3.4.1 Solution Enumeration with Modified DPLL

The DPLL algorithm with simple backtracking presented in chapter 2 inverts decision variables when it runs into a contradiction. It is possible to use the same mechanism to implement solution enumeration. The DPLL-based solver with solution enumeration that we present uses an additional stack in order to prevent itself from reproducing identical solutions. After each solution is found, the solver continues the search for satisfiable solutions by moving to a different part of the search space. The solver terminates when the search space is exhausted.

The enumeration solver in GridSAT uses an additional stack called the *secondary stack*. The function of this stack is to track the state of each decision variable on each level of the decision stack. Thus the size of the secondary stack is equal to the size of the decision stack and is at most equal to the number of variables in the original SAT problem. Each level in the secondary stack has one entry which tracks the state of the decision variable at the corresponding level in the decision stack. An entry in the secondary stack is a flag which indicates whether the decision variable is either *flipped* or *not-flipped*. A flipped variable is one whose value has been inverted after the initial assignment was fully investigated. In the following description we also use the equivalent pair of terms *inverted* and *non-inverted* Initially all entries on the secondary stack are marked as not flipped.

The enumeration process does not impose any restrictions on the normal execution of the DPLL algorithm. The algorithm start execution normally by speculatively assigning values in its quest to find a solution. The solver adds new decisions and backtracks in the usual fashion. The secondary stack shrinks and expands mirroring the decision stack. When a solution is found the current variable assignments are saved in a repository external to the solver process.

After the assignment representing the solution is saved, both stacks are modified. First, the decision variable in the highest level of the decision stack is inverted. Second, the corresponding flag entry in the secondary stack is marked as *flipped*. The solver continues by clearing the highest decision level and inverting the value of the same decision variable. After both stacks are updated, the sequential solver proceeds as usual by making more speculative decisions, augmenting the decision stack and backtracking when a conflict is encountered leading to a reduction in the decision stack. When the decision stack shrinks because of backtracking all states in the secondary stack above the current decision level lose their logical significance and are cleared by marking all of them as non-inverted.

When a solution is encountered and the current decision level in the secondary stack is marked as inverted, the solver proceeds by removing the highest decision levels and backtracking to a level where the decision variable on the secondary stack is not flipped. When such a level is found before reaching the second decision level, the solver marks that level as flipped and continues by assuming the inverted value at the same decision level. For example, in figure 3.5 a solution was encountered at level four as shown on the left side of the figure. The right hand side shows how the decision and secondary stacks are modified. The solver backtracks to level 2 since it is the first non-flipped entry encountered on the secondary stack. This entry is marked with an **O** (not-flipped) before the solution is found. The same entry is marked with an **X** (flipped) after the solution is found. Notice that entries in the secondary stack (level 3 and 4) below the new decision level (level 2) are cleared and marked as not-flipped after the solution is found. Also the variable at the new decision level V_7 is flipped in the modified decision stack to $\sim V_7$. After updating both stacks, the solver proceeds to explore the rest of the search space.

If the solver backtracks to the second decision level, then the solver has finished sweeping the branch of the search space which assumes the current value of the decision variable at this level. Therefore, the solver can assume the opposite value of this variable for the remaining search space. Thus, the solver backtracks to the first decision level and augments this level with the inverted value of the decision variable previously found at the second decision level. The solver then proceeds by searching for implications produced by the newly assumed values.

The secondary stack is used as an additional mechanism to restrict the search space after a solution is found. In addition, the new extension to the basic algorithm does not restrict the efficient sequential algorithm in any fashion. The solver continues to navigate and prune the search space as before. The role of the secondary stack is to prevent the solver from reproducing the same solutions unnecessarily.

3.4.2 Parallel Solver with Solution Enumeration

This section describes the process used to adopt solution enumeration in the context of a parallel solver. Deploying parallel solvers with solution enumeration only requires minor modifications to the version presented in section 3. The role of the parallel solver infrastructure is to collect all the solutions in a repository.

In the parallel version described in section 3.2, each solver is given an initial decision stack and a clause database. In contrast,in the parallel solver with solution enumeration, each client is given a clause database, a decision stack and an additional secondary stack. In order to illustrate how the secondary stack is split in the case of a parallel solver with solution enumeration, we use the example in figure 3.6. Before splitting, the original client A has both a secondary stack and a decision stack. Client (B) gets the exact same decision stack as previously described in the splitting process of section 3.2. The new client (B), also, receives a totally blank secondary stack. Client (A), however, might get a decision stack with more variables on the first decision level and smaller number of levels. Initially, the first decision level is augmented by adding all variables from the second decision level. In addition, this level is padded with any sequence of levels with flipped decision variables starting at level three on the original stack. These variables are added for the same reason that all variables in the second level are added. When the flip variable is given a final value, all these variables are automatically assigned the current value because all other options have already been investigated. Also, the new client (A) will receive the same original secondary stack except that all levels deleted from the decision stack are also deleted from the secondary stack.

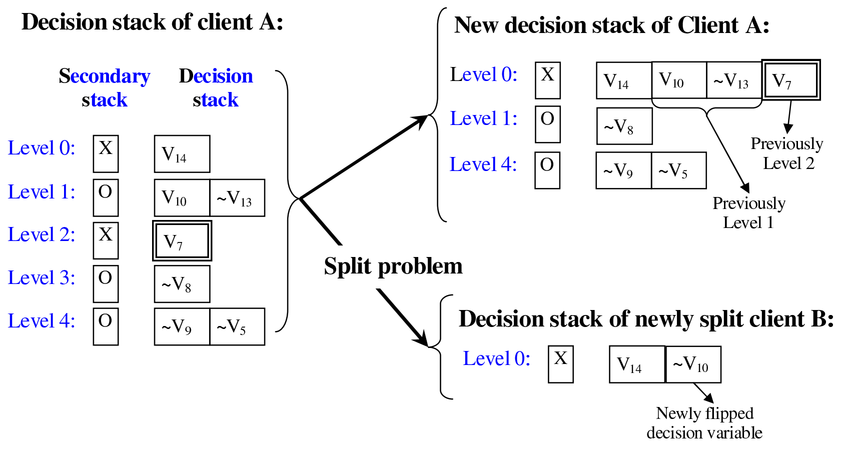


Figure 3.6: Decision stack and secondary stack modification during splitting when solution enumeration is used. In the secondary stack X and O stand for flipped and not flipped respectively.

In figure 3.6, we show an example of problem splitting in a solution enumerating parallel solver. The initial client A, has a secondary and a decision stack. In the secondary stack, the first and second levels are marked as flipped. Since the first level only contains variables with final values, it is always flipped. After splitting, the client (A) gets a new decision stack which includes three additional variables. Two of these variables (V_{10} and $\sim V_{13}$) come from the second decision level. The third variable (V_7) comes from the third decision level because it is marked as flipped. In this case, the third level does not have any other levels flipped immediately after it. Thus no other variables are added. The new secondary stack for client A has two levels (1 and 2) deleted since these levels have been removed from the decision stack. On the other hand, the newly created client (B), gets only an additional variable (V_{10}) on the first decision level. Also client (B) gets a secondary stack which also contains one level.

This method of solution enumeration will not produce redundant solutions. In a parallel solver, each client starts from a distinct initial decision stack as described earlier. If any client finds a solution, the initial decision stack will be a subset of that solution. Therefore, it is guaranteed that no two clients will produce the same solution since all clients start from distinct initial decision stacks. Furthermore, no client will produce the same solution more than once because the decision stack is different for each iteration of the DPLL algorithm. Therefore, the above algorithm will produce a set of distinct solutions.

Chapter 4

Programming Methodology and Application Architecture

In this chapter we describe the design and implementation of the GridSAT application. The design of the GridSAT application has three main goals. The first goal is to enable a parallel SAT algorithm that permits the use of a variable number of resources depending on the specific *needs* of given SAT instance. The second goal is to make efficient use of available resources. The final goal is to make GridSAT adapt to variations in the availability and composition of the resource pool.

These goals can be achieved using an efficient implementation which is tailored to the overall characteristics of the application. We start by describing the main features of the satisfiability solver. Then, we describe the main design components and their interaction. Finally, we discuss the programming methodology used to realize our GridSAT application.

4.1 Execution Model

Most traditional parallel applications make resource selection decisions (i.e. resource scheduling) based on a performance model. This model gives a precise prediction for the cost of execution of a particular task in a particular environment. Under such conditions, it is possible to predict the time duration needed to execute an entire parallel job. It is also possible to quantify the effect of the size and type of available resources on the turnaround time of the application.

The satisfiability problem, however, does not have an execution model that would allow for time or resource prediction. This is a major difficulty facing the SAT solver community. It is not currently possible to predict how much time and resources are needed to solve any SAT problem. In addition, some satisfiability problems can be *easy* because they can be solved quickly with few resources. Some other problems are *hard* or *complex* because they require a long time and many resources to solve them. In fact, due to the interaction between learning heuristics and the data dependent nature of SAT problems, a SAT problem instance may be perceived differently by two SAT solvers. For example, a problems that is deemed easy by one solver may be found difficult by other solvers depending on the heuristics each employs. Thus, it is not currently possible to determine the resource and time requirements for a given satisfiability problem just by inspecting its logical formula.

There are, however, some heuristics that can be employed to detect when a SAT solver is making progress. One heuristic examines internal solver state to estimate both the rate at which the solver is pruning the search space and the rate at which it is exhausting memory and then extrapolates time-to-solution from these rates. If the ratio of the speed with which the solver prunes the search space to the rate at which it is consuming memory is low, the problem is perceived as being "hard" according to this heuristic. For a hard problem, a learning solver will either run out of memory and terminate, or prematurely discard and then subsequently relearn clauses to avoid memory exhaustion and, because it must run longer, may eventually incur a user timeout.

The execution model we adopt for GridSAT uses incremental resource admittance and release. Since the resource requirements of a given SAT problem are unknown, GridSAT starts by using a small number of resources. This strategy incurs low parallelization overhead for easy problems that can be solved quickly with a few resources. Thus easy problems can be solved in a time duration comparable to that of a sequential solver. For harder problems, a timeout interval is used after which a given resource can split its search space with another one. The timeout interval is used to amortize the splitting cost and to avoid the ping-pong effect introduced in section 3.2.5. The timeout interval is defined in function of the overhead of the splitting process as detailed later in section 4.3.2.

As the execution progresses, new resources are added to participate in solving the current SAT instance. Simultaneously, those resources that are done investigating their assigned portion of the search space are release immediately. In this manner, the resource pool used by the GridSAT solver can expand and shrink based on the SAT problem. If the problem is hard the set of resources will expand. On the other hand, if the problem is perceived to be easy the number of resources in use can decrease. Actually, the same problem can exhibit different phases where it is at times perceived by the solver as hard and at other times as easy.

Finally, the GridSAT solver terminates when all sub-problems have been solved or one of the clients finds a satisfying assignment. In the latter case, the client that finds the satisfying assignment sends its stack to the master. The master verifies that the set of truth assignments it received does indeed satisfy all clauses in the initial problem. Most solvers in the literature are evaluated based on the time the first satisfiable instance is found. But there are cases [56] where knowing all satisfiable instances is helpful. GridSAT can also enumerate all the instances where a problem is satisfiable. In both cases, when the master determines that the problem is solved, it sends a message to all clients requesting them to terminate.

4.2 Application Characteristics

The GridSAT application is different from many other high-performance computing applications in terms of programming model and resource usage. Traditional higher-performance applications use the Single-Program-Multiple-Data (SPMD) [57] or bulk-synchronous parallel (BSP) model [110]. In general, these programming models are characterized by a set of alternating steps involving computation and communication. In addition, the computation and communication intervals do not overlap. The communication steps are followed by synchronization barriers and enable the various components of the application to exchange information. From the resource usage perspective, these applications use a predetermined set of compute resources throughout their execution. Actually one of the motivations for using BSP is the ability to predict the execution cost (i.e. time) of the entire program based on the source code and a few parameters characterizing the execution environment.

Based on the execution model described it the previous section, our application differs in much of the above aspects. The GridSAT application has variable resource requirements depending on the problem instance. In general, the number of resources and duration of use of those resources cannot be predicted for satisfiability instances. In fact, the set of active resources that are assigned parts of the search space during runtime is dynamic. On one hand, resources are added each time the problem is split. On the other hand, resources are released immediately after a subproblem is solved. At any given instant, the application can simultaneously acquire new resources and release other unneeded resources. Moreover, the application components communicate with each other to share intermediate results as soon as they are produced. These results are asynchronously used by all the receiving clients.

Therefore, all the GridSAT components are event driven and events are produced and consumed asynchronously. The solver components, for instance, can simultaneously perform communication and computation. All application modules are designed and implemented to allow for efficient management and handling of these events.

Dynamic resource usage can potentially solve a large set of satisfiability problems an efficient manner. Solving “hard” satisfiability problems introduces further challenges. For “hard” problems, a small number of resources would be exhausted in a relatively short time. While solving the hard problems, available CPU and memory resources can become saturated, thus additional resources are required. Therefore, we need to use all computational resources at our disposal, in order to render the solution of the hardest problems more plausible. The set of available resources varies from desktop machines, to small-size clusters, to supercomputers. This collection of resources is heterogeneous in terms of hardware, operating systems and resource management software. This heterogeneity represents a further challenge in the deployment of the application.

The application characteristics described above are representative of a true computational grid application. As more “power” is added to the grid that GridSAT is deployed on, GridSAT’s flexible design enables it to efficiently use the available resources. At the same time, fluctuations in available power are tolerated automatically so that the overall application remains maximally efficient while it is executing. Thus, GridSAT is designed to be one of the first programs to realize the vision of grid computing originally articulated in [43] and to demonstrate this capability by generating new domain science. Moreover, these characteristics are not unique to GridSAT. Other branch-and-bound or master/worker applications can benefit from similar use of computational resources.

4.3 Application Architecture

GridSAT is implemented as a special form of the master/client model where individual clients communicate directly and share clauses. There are two main types of processes: the master and the client. In the next sections we describe the design of both of these components.

4.3.1 Design of the Client Process

The client process is designed to be a “cooperative” satisfiability solver. The main component of the client is the sequential SAT solver. This solver is modified in order to enable it to interact with other components of the parallel satisfiability solver. The client process is designed as a threaded program with two threads. The main thread is the SAT solver and the helper thread is the communication thread.

The solver thread is perfoems four main functions. These functions are:

- Problem Splitting: This function allows the solver to split its search space and send the new sub-problem to another client. This procedure is expensive because it requires that the entire database be analysis twice. The set of relevant clauses for the new search space has to be determined once for the new client and once for the current solver. Also this procedure involves sending the largest message consisting of a large part of the clause database to a new client. In order to avoid making an extra copy of the clauses the solver is stopped momentarily as the clauses to be transmitted are identified.

- Memory management: The solver is designed to use as little memory as possible. Thus, the clause database is allowed to expand till the host memory becomes scarce. At this point the clause database is reduced by shedding some learned clauses if possible. By reducing its memory usage the client can allow other programs to use the host while avoiding overloading the hosts memory or other OS specific policies such as the "Out-Of-Memory killer"in Linux.

- Clause Sharing: Each client saves generated clauses below a certain configurable size. These clauses are periodically shared with other clients.

- Clause Merging: The client continuously receives clauses from other clients. These clauses are saved temporarily in a buffer and then merged based on one of the merging policied used as detailed in section 2.3.

The communication thread is responsible for exchanging messages with the master process as well as other clients. The goal of this thread is to limit the interruption of the main solver thread. This thread is responsible for handling three types of messages. The first message type requires immediate action from the solver thread. Such examples include messages that require the solver to exit after a solution has been encountered. The second type of messages can be delayed until the solver thread is ready to handle their content. For example, received clauses are buffered before they are merged by the solver into the clause database. The third type of messages includes periodic messages such as heartbeat signals that indicate which resources are still active.

4.3.2 Design of the Master Process

Each GridSAT client keeps track of its own progress while solving sub-problem it was assigned. The process which tracks the status of the entire SAT problems is the *master*. The master process is the only process the user interacts with. After its instantiation, the master is responsible for assigning sub-problems to clients and deciding when the parallel solver terminates.

A general architecture of the master process is shown in figure 4.2. The master also uses external services which are shown in the "clouds." The master consists of four main components: the resource manager, the client manager, the scheduler and the checkpoint server.

The resource manager loads resource information from one or more Grid information systems such as Globus MDS [31] and the NWS [117, 103, 116]. The scheduler, on the other hand, is responsible for coordinating the interactions between all the components. In addition, it handles interactions with external resources and monitors them to detect failures. The resource manager is aware of the different types of resources. Thus, only one GridSAT process per host is launched for shared resources. Also, the resource manager launches one job at the start of the execution for batch systems. Additional jobs can be manually submitted and GridSAT can use their resources when they become available. Actually, the client manager will use any additional clients launched from newly available resources or previously submitted batch jobs. It is the role of the client manager to maintain the list of active clients and monitor their progress.

The GridSAT scheduler is the focal point of the master process and is responsible for coordinating the rest of the components. It is also responsible for launching the clients. The scheduler uses a progressive scheme for acquiring resources and adding them to the resource pool. Also resources that are no longer performing a task on behalf of GridSAT are released immediately when possible. The reason for this approach is the variability and unpredictability of resource usage for a particular SAT problem.

A typical execution will start by launching the master. The master will examine the problem to find any obvious variable assignments and remove any inconsequential clauses. Some problems might be solved at this stage because of an easily detectable conflict. After this stage, the master requests the resource list available from deployed grid services such as the Globus MDS [31] and NWS [117]. The master may also use a configuration file to obtain resource information. After this step, the scheduler immediately submits any batch jobs to their respective queues. When a remote client starts running it contacts the client manager and registers with it. The scheduler ranks the set of available clients based on their processing power and available memory as provided by the NWS [117, 103]. Static values for these resource parameters can be used when GridSAT is configured without NWS or the Globus MDS.

The GridSAT scheduler uses the first available client to immediately start solving the problem. Each client records the time it took to receive the problem data. Clients also monitor their memory usage. The decision for splitting a problem is made locally by the client and not by a centralized scheduler. A client decides to split its subproblem when its memory usage exceeds a certain limit or after running for a specified period of time. This time period is determined as two times the duration of the communication period the client used to obtain the problem data. Using this method, the scheduler allows for computation time to offset the communication overhead. The clients, therefore, do not spend most of their time splitting instead of doing useful computation, thus avoiding the "ping-pong" effect described in section 3.2.5. When a client wants to split its subproblem, it notifies the master.

The GridSAT master also uses a threaded execution model similar to the client. The master has two main threads: the communication thread is responsible for receiving messages and storing them in an event queue, while the scheduler thread removes events from the event queue and services them one at a time.

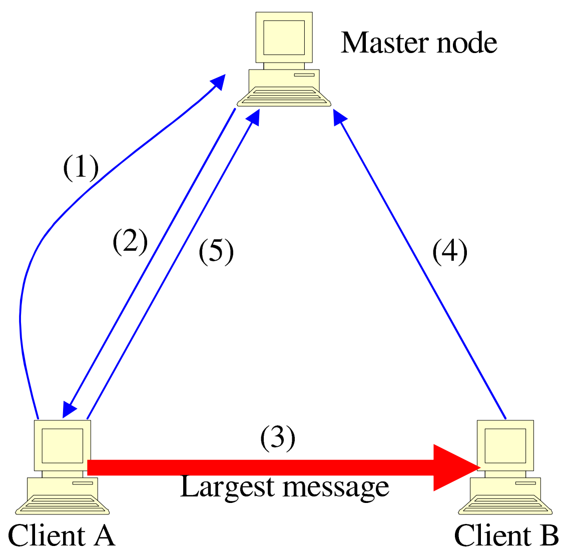


Figure 4.1: Message exchange between master and client during the splitting process

4.3.3 The Splitting Process

One of the most important operations in GridSAT is the splitting process. The logical concept is described in section 3.2. The implementation of this process requires the cooperation of three components: the master, the splitting client and an *idle* client. The *idle* client is a process that was not previously assigned a sub-problem to investigate. The splitting is realized in a way as to minimize communication costs.

Figure 4.1 shows a step-by-step description of the splitting process. Client A that has presumably been solving a sub-problem, has detected that it needs to split its search space. Client A then notifies the master using message (1). Upon receiving this message, the master selects the highest ranked client and includes it in message (2) which it sends to client A. Using the information in message (2), client A determines which of its peers it will split the problem with. Client A then proceeds to communicate directly with client B by sending it message (3). This message is very large and varies in size from 10 KB to 500 MB. By using direct peer-to-peer communication the overall communication overhead is reduced. When the splitting is successfully completed, both clients independently alert the master using messages (4) and (5). In Message (4), client A sends new stacks for both clients A and B. Each stack is used as a checkpoint for its respective client. Both messages are used so that GridSAT can recover gracefully if one or both clients fail during the splitting procedure. Also if only one of the clients fail, then only that client will be restarted because acknowledgements (4) and (5) are received separately.

Message (3) above allows the transfer of a newly created sub-problem to the idle client. This message is the largest message and contains three different parts:

- The assignment stack: It is the smallest part and is in the order of the number of variables.

- The set of original problem clauses: This could be as large as the initial problem file
- The database of learned clauses: It is the largest component and is 100s of MegaBytes in size.

4.3.4 Reducing Communication Overhead

GridSAT reduces the communication overhead of the solver in two ways. First, problem files are copied only once when several hosts share a common file system. Therefore, split messages targeting the same set of hosts will be smaller since they will not include the second part of message (3) discussed above. The second modification makes it possible for the new client to proceed with its computations immediately after it receives the assignment stack and load the problem file from the shared file system. Since learned clauses contain redundant information, then they are not required to start solving the new sub-problem. Therefore they are sent in a separate message. This message takes a long time to transfer, and the new clauses will be merged as they are received using the clause merging algorithms discussed earlier. Using these techniques, the new client (client B in figure 4.1) will not have to idly wait for the entire message to arrive before starting solving the newly assigned sub-problem. The old client (client A in figure 4.1) still waits because the clause database is locked until the transfer is completed. Making an additional copy of the clause database in order to prevent the old client from stalling is not practical because the size of the clause database is very large and there is not sufficient memory to hold a separate copy. The old client waits and does not proceed until the clause database destined for the new client is transferred. Transferring these clauses to the new client is essential to the efficiency of the solver. Eliminating this transfer would slow the solver significantly.

4.3.5 Failure Recovery and Checkpointing System

A computational grid environment resource is composed of numerous resources and network components. Thus the probability that any one element of the computational grid fails is much higher. As a result failures are more frequent in such an environment. Therefore a grid application has to be able to recover from such failures. There are two components to failure recovery. First, the grid application should establish a mechanism for detecting the failure of remote components. GridSAT uses heartbeat messages to decide when a remote solver has failed. Second, the grid application should be able to restart with minimal work loss when failures occur. The current version of GridSAT uses checkpointing to recover from such failures. Each checkpoint belongs to one remote solver and represents a SAT sub-problem that can be restarted when combined with the initial SAT problem. GridSAT can use two types of checkpoints:

- Light checkpoints: This method requires little storage space and communication overhead. Only the top level of the assignment is recorded in the checkpoint for each client. In this case checkpoints for a client will be updated only when more variables are added to the first decision level. This form of checkpointing records the most important advances of the SAT solver while using little storage and communication overhead.

- Heavy checkpoints: In addition to the light checkpoint data, heavy checkpoints save all newly learned clauses. It is also possible to save the top levels of the decision stack in order to reconstruct the exact decision levels after restart. These checkpoints can be saved at regular time intervals in addition to the instances when the top level is augmented. These checkpoints require more storage and incur higher communication overhead compared to the previous form. However, they are more effective when the solver does not make progress which results in modifying the first decision level. In this case the progress is evident only at higher levels of the decision stack.

The master stores and updates the checkpoints as they are received from the clients. The checkpoints can be stored either on a local file system or in a distributed fashion using IBP [75]. Idle clients are assigned new sub-problems either through splitting or from saved checkpoints. Sometimes the number of checkpoints exceeds the number of active clients. This happens when a large number of previously active clients terminate leaving behind their checkpoints. In this case the scheduler keeps a list of checkpoints and assigns them to newly created clients or those that have just finished solving their own sub-problem. Idle clients are assigned problems through splitting only after all checkpointed sub-problems are assigned to active clients.

When the master fails, GridSAT can recover by simply re-instantiating the master process on another machine if necessary. If checkpoints are available, the new master process can use them to recover pre-failure state. Also a user could cause an *intentional* failure by halting the master while it is solving a problem in order to start another problem for example. The user can later resume solving the previous problem using the saved set of checkpoints.

4.3.6 Work Backlog

The GridSAT execution model allows for the application to generate new tasks depending on the number of resources available. As time progresses, more and more tasks are produced as more clients request that their problem be split. The master records these requests and keeps a backlog so that at a later time when a a resource becomes idle, the master can choose a client that has requested a split, and allow that split to proceed. The master splits clients that have been running the longest on the same subproblem. This strategy gives more resources to those parts of the search space that take the longest.

At a global level, the master ensures that most of the clients are doing useful work. This is accomplished by setting a limit on the number of clients splitting simultaneously. The maximum number of clients involved in splitting is set as a configurable parameter. The effect of this strategy is to dynamically lengthen the splitting timeout interval when the number of splitting clients grows large. Therefore, only a small portion of clients will be splitting simultaneously at any instant. As a result, even if the resource pool is very large, most resources are guaranteed to be doing useful work instead of splitting.

4.4 Multiple Site Scheduling and Migration

GridSAT processes communicate as peers during problem splitting. The optimization presented above reduce communication overhead due to master/worker communication. Most of the remaining overhead is due to peer-to-peer messages. Therefore, more efficient problem splitting will improve overall solver efficiency. More efficient problem splitting could be accomplished when clients belong to a pool of well connected resources. Such pools of resources usually become available when new batch jobs reach the head of their waiting queue and start running. GridSAT migrates problems from dispersed nodes to processes that are part of a batch job.

Efficient Use of Batch Jobs

Existing computational grids contain two types of resources. The first type includes shared resources that allow many users to execute jobs simultaneously on the same hosts. The second type comprises time shared resources such as supercomputing facilities [22, 106] and collections of grid resources such as Condor [29, 104]. In these resources, a batch scheduler gives a user exclusive access to a subset of hosts for a particular time period. Users in these environments are given a budget (i.e. a quota of CPU-hours) to use. Since this is valuable time, it is important from the user's perspective to use it effectively. The scheduler bills the user and deducts from his budget an amount proportional to the total time and number of nodes his job consumes. The user is billed for the time used and not the time he initially requested. Thus if a job terminates early the user is only billed for the time during which his job actually ran. From a user's perspective, the goal is to minimize the cumulative idle time for all nodes during a batch job execution.

In traditional parallel applications, such as MPI [69] programs, the number of processes spawned is sufficient to ensure that all nodes have a slice of the work assigned to them during the entire duration of the execution. All nodes start and stop execution simultaneously. This scenario leads to an efficient use of the batch jobs. GridSAT is not a traditional parallel application. In the case of GridSAT, the number of jobs (i.e. sub-problems) varies during execution. Actually, when a new large batch job becomes available the number of workers might be much larger than the number of available sub-problems. The goal of GridSAT is to make good use of the newly available and valuable processing power. It is possible to immediately split a sufficient number of sub-problems. This will lead to more efficient use of batch jobs but it may also affect negatively the solver's performance.

If GridSAT, however, waits till enough problems split to populate all the batch nodes it may lead to an inefficient use of super-computing nodes.

In GridSAT, initial batch job requests are large with a high number of nodes and long durations. This leads to a long waiting period in the scheduler's batch queue. Thus if a job is not solved after this long waiting period than it most probably is a hard problem. Thus batch jobs are only used when the problem is hard. When a batch job starts execution, GridSAT uses problem migration to achieve more efficient use of batch nodes. Numerous remote GridSAT nodes will migrate immediately to occupy batch nodes. At this point, splitting happens at higher rates because super-computing nodes are linked by a high performance network. Also the number of active nodes (i.e. those with sub-problems) will increase exponentially. This happens because the number of new sub-problems is increased in proportion to the number of existing active solvers. Therefore, problem migration leads to a more efficient use of batch jobs.

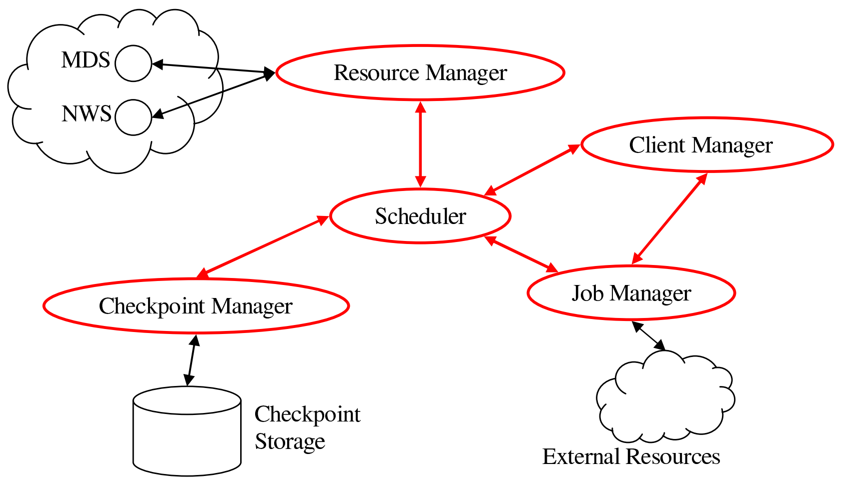


Figure 4.2: GridSAT components and their internal and external interactions. The external components and systems that GridSAT uses, such as the Globus MDS and the NWS, are shown in clouds.

4.5 Concurrently Running Multiple GridSAT Instances

In a computational grid environment resources may be shared by multiple applications. As these applications may have conflicting needs they tend to stress the different components of the computational infrastructure. For example, the memory and CPU in computational nodes may be exhausted by the demands of the grid applications. Similarly networks can be saturated as different applications communicate internally between their distributed components. Under such conditions the resources may be overloaded and become unresponsive. Such results are not desirable and may be disruptive to those applications that wish to make use of a given resource. In order to increase efficiency of the resources and the Grid applications, these applications should be designed to avoid overloading the computational infrastructure.

A well designed grid application should avoid depleting a computational resource to the point where it becomes nonfunctional. A grid application should accomplish this goal in order to curb any disruptive effects its activity could have on the computational environment. This goal could be accomplished by monitoring certain vital characteristics and taking appropriate measures when possible to alleviate any problematic symptoms. A well known example for sharing common media is the TCP backoff mechanism [94, 113] that allows several senders to fairly share a common communication medium. In addition, an application should tolerate the failures of resources because such failures are not always avoidable.

GridSAT is a resource intensive application in terms of host memory and CPU as well as network load. Running multiple GridSAT instances simultaneously will further load a given set of resources. There are many possible scenarios where multiple GridSAT instances can be deployed simultaneously. For example, a user might want to solve many SAT instances but it is not clear whether the execution should be sequential or simultaneous in order to solve as many of the problems as possible. Also many users may share a common set of computational nodes for the purpose of solving SAT problems.

One possible solution is to schedule resources using a batch system where each GridSAT instance will be given exclusive access to a subset of the nodes for a given time period. This solution is a common method used for allocating computational resources because it limits the effects of one application on another. Such a solution,however, is not totally effective since applications still share the network and other services. In addition, if the SAT instance being solved does not use all the allocated resources many nodes will remain idle. Thus valuable computational power is wasted because it idle nodes are not available for use by other applications.

Another possible solution would be to allow the GridSAT instances to execute on the resources simultaneously. This solution is more suitable for GridSAT since the number of resources used by the application varies. Thus executing more than one GridSAT instance would make it possible for those resources that are idle to be used to solve some other SAT instance.

In the next two sections we describe the mechanisms used by GridSAT to enable multiple instances of the applications to effectively share a set of resources.

4.5.1 Adaptive Memory Allocation

The size of memory available to a solver is an important factor in determining its efficiency. A GridSAT solver has little utility for resources with heavily loaded memory. When a solver has little memory available it cannot store many learned clauses, thus limiting its effectiveness. In this case splitting the problem with a memory starved resources is not likely to improve time to solution. At the same time, these starved resources will experience additional depletion of whatever little available memory they still have. This may in turn adversely effect the performance of other applications already executing on that resource. Thus, each GridSAT client is configured to only use resources when a minimum memory size is available. Actually, the client ensures that a configurable memory size ($50MBs$ by default) is free after the instantiation of the client. Allowing for free memory allows other small programs such as Linux shells and commands to be initiated so that the resource is still responsive.

The GridSAT solver starts by allocating a clause database of minimal size. As the solver progresses it increases the size of the clause database incrementally to allow for more learned clauses to be stored. The size of the database cannot, however, grow indefinitely. The solver is limited by the memory available on the host. The size of the free memory is dynamic and depends on how much memory the applications running currently on the host are using.

Each client implements a local memory sensor based on NWS memory sensors. This enables the client to get instant information about the size of free memory on the host. When the solver detects that the amount of memory is far lower than a certain threshold ($50MBs$ by default) it shrinks its clause database so that the size of free memory is not below the desired limit. The client can later expand its memory usage if more memory is available.

At the start of the execution clients can use all the memory available. However, when a new client wishes to share the same resource, the clients already executing on that resource shrink their memory usage. If enough memory is freed the new client can instantiate successfully. However, in some cases not enough memory will be available so the new client will not be able to use that resource. The overall effect is that only a maximum number of clients execute per host. At the same time each of the clients is allocated enough memory to make progress. This strategy avoids the scenario where many clients share the resource but none of them is making significant progress.

If the number of GridSAT instances is large enough the resources may be saturated. In this case all resources are busy and no additional sub-problems can be successfully split. Thus each GridSAT instance temporarily suspends splitting. This decision is made in a distributed manner without a central scheduler or GridSAT masters communicating with each other.

4.5.2 Resource Scheduling

Each GridSAT instance keeps track of its own resource usage and will not launch more than one client per resource. GridSAT uses NWS to receive information about CPU and memory usage of resources. The default configuration of NWS sensors updates resource parameters every ten seconds. This period represents the time interval during which resource information and performance characteristics might change without the application becoming aware of it. We call this period the Resource Information Update Period (RIUP).

During this period the various GridSAT schedulers might have to make many decisions concurrently. The characteristic of a given resource (CPU usage and memory available) may change within the RIUP because of resource allocation decisions and activity by other application. In fact, many GridSAT instances can initiate many new clients within a few seconds because each client instantiation only takes a few milliseconds. Thus the monitoring information could become stale during each time interval before the sensors update their data.

In the initial implementation of GridSAT, we did not take any precautions to ensure that running many GridSAT instances will not result in negative effects. All GridSAT instances base their resource allocation on a common scheduling policy and resource information obtained from NWS. The GridSAT scheduler ranks resources based on their available CPU and memory. Thus when many GridSAT instances start selecting least loaded resources using stale data they will make similar decisions. As shown later in the results section 5.5.1, this naive strategy causes a "herding effect" where all masters select the same resource for starting new clients. This has a negative effect on the performance of the solvers since they are overloading one resource while other resources are unused. For example, at the start many resources are free so the different GridSAT instances will make the same selection decision while there are other equivalent options. The herding effect might also occur if a few resources are selected for launching new clients compared to the number of GridSAT instances requesting new clients at any one moment. Situations like these make the herding effect even more pronounced.

After making this observation our goal was to avoid the herding effect while making little modification to the GridSAT scheduler. An additional challenge was to provide a single implementation that could be used by all GridSAT instances. Also we wanted the solution to avoid explicit synchronization between the GridSAT instances. Explicit synchronization can be implemented through direct communication between multiple GridSAT instances or through a global entity. Avoiding explicit synchronization and information sharing is important for two main reasons. The first one is scalability because direct communication or a global synchronization approach would have a high overhead and would scale to a large number of application instances. Second, using explicit synchronization may not always be possible because it might be adversely affected by other grid applications that do not use the same solution.

The solution GridSAT adopts solves the herding effect problem using a two step approach. First, the resource selection chooses all resources that are within 10% of the highest ranked resource. The specific resource used to start the new client is selected randomly from this set. This method ensures that the least loaded resources are used to launch new clients. At the same time when the difference between resources is not significant it allows the master to choose randomly amongst those resources.

This solution minimizes the chances of occurrence of herding but does not completely avoid it. The “herding effect” is still likely to occur when many resources are similarly loaded. Such a condition exists for example, at the start of execution where many resources are free while many GridSAT instances are frequently expanding their resource usage. As shown in the results section 5.5.1, the herding effect was significantly reduced. As time progresses, GridSAT instances will request new resources less often and at different moments thus reducing the possibility of a resource being selected by many masters at once. Another condition that GridSAT tries to address is when a large number of GridSAT instances are competing for few resources. In this case, we cannot completely avoid the herding effect. The solution we adopt allows the GridSAT instances to compete for the resource and only some of them will be able to successfully start a new client. The other GridSAT instances react by incrementally waiting for longer intervals before trying again to start a new client. The result is that when all resources are busy GridSAT instances in need of more resources will wait and not interfere with other running applications.

4.6 Programming Methodology

A major challenge before implementing the various application components was to develop an implementation strategy. The final implementation aims at using all the available grid resources efficiently while dynamically adjusting to the application behavior and resource needs.

Given the resource usage patterns of GridSAT, which are typical for a true Grid application, we had to choose an implementation strategy that would satisfy these requirements. There are several technology choices to select for the implementation of the application. Such options include, among others, MPI [69], Globus [42], vanilla Web Services [118] and later improvements such as WSRF [73]. These technologies represent low level programming interfaces. Other projects such as GAT [13] aim to provide a simplified and unified interface to other Grid middleware.

According to our experience with GridSAT we have learned that a successful implementation technology should allow for three pivotal capabilities: dynamic resource pool management, error detection/reporting, and universal deployment.

The first capability is to allow the use of a dynamic resource pool. This feature, for example, was not available in MPI-1 which did not allow for dynamic Communicators. MPI-2 has introduced extensions to allow for dynamic creation and destruction of communicators. Globus and Web services also allow for a dynamic set of resources.

The second capability is error detection and reporting. Since GridSAT runs for extended periods of time using a set of geographically distributed resources, network and resource failures are more frequent. Therefore in order to implement this application we need a means to detection of these errors. From the perspective of the application, the distinction between resource and network failures is not important. It suffices for the application to obtain a feedback if a certain operation is not successful after a certain time period.

Error detection and recovery are very important because in our experience all resources experience a failure at some point. Even those resources that are professionally maintained can become unresponsive from the application's perspective. Those resources that do not experience hardware and software failures usually have routine preventive maintenance periods or a combination of software and hardware upgrades. From the point of view of the application these are "scheduled" or "anticipated" failures. Without rigorous error handling the application would not be able to run for extended periods as shown later in the results section.

Different programming tools provide some form of error handling. MPI-1 allows for error handling in a limited scope which is expanded further in MPI-2. Globus GRAM [44] allows for error handling and call-back functions for job management. In Web Services, WS-Notification [50], WS-BaseFaults [101] and related standards could be used to provide this functionality. Other high level projects such as GAT [13] mirror, albeit in a simplified manner, the error detection capability of the underlying middleware.

The desirable error handling policy for our application is to provide a time period for some actions after which some form of error handling should be performed. Sometimes if an action fails, then all that is needed is to retry it. In other cases, it is assumed that the resource (or the connecting network) has failed. This form of error handling is not available for the grid technologies mentioned above; therefore it needs to be implemented at the application level.

The last desirable capability for a suitable grid technology is universal deployment. This is not only a characteristic of the technology but of the computational environment as well. A widely deployed technology is advantageous because it reduces the development overhead since one version can be deployed on all available resources. In our experience, there was no grid technology that was universally adopted and deployed enabling us to combine all computational resources at our disposal. Thus a multi-infrastructure approach such as EveryWare [115, 114] was necessary.

Furthermore, in order to deploy our application over a large set of resources,we had to interface with many types of resource managers. For example, resources could be managed by one of many batch schedulers like PBS and Condor [104] or could be simply shared. Our goal was to use all these resources simultaneously regardless of which systems managed them. This is accomplished by determining a general job description that can be instantiated differently using specific launchers for each resource manager. For instance, shared resources can be accessed directly using SSH. Batch systems, however, are accessed by submitting a batch script with syntax tailored to the scheduler used. Whenever Globus is deployed we use it to launch and monitor job submissions.

4.6.1 GridSAT Implementations

We believe that many of these technologies could be used to develop GridSAT. In fact, we have developed a previous versions of GridSAT called GrADSAT [27] (note the "A" in the spelling) using *GrADSoft*. GrADSoft is a set of programming abstractions where the baseline grid infrastructure is provided by Globus and the NWS. GrADSoft is part of the **Gr**id **A**pplication **D**evelopment **S**oftware (GrADS) project [18, 51] which is a comprehensive research effort studying grid programming tools and application development. To facilitate experimental application research and testing, the project maintains a nationally distributed grid of resources for use as a production testbed. Since the GrADS tools were universally deployed on this testbed we were able to deploy our application with little effort on the entire testbed.

The current version of GriDSAT uses EveryWare [115, 114] a very portable communication library. EveryWare has been designed explicitly to manage the heterogeneity and dynamism inherent in grid resource environments. EveryWare can be easily deployed as library on all the resources. In addition, all communication calls use a timeout argument, as desired, for error detection.

The resource management system interfaces with resources that use batch systems as well as desktop machines that are accessible through SSH. All resource related operations have been implemented to allow for a specific timeout. If the resource is not responsive after the timeout period expires, then the resource is considered unreachable.

Chapter 5

Experimental Results

In this chapter we evaluate the performance of the GridSAT solver from five different perspectives. First we study the effect of changing the fraction of clauses shared during the split on the GridSAT performance. Second, we compare the merging strategies that were presented in section 3.3. These two sets of experiments help determine what parameters are used to increase the solver's efficiency. Third, we present empirical evidence that the GridSAT solver outperforms the original sequential solver zChaff. Fourth, we showcase the ability of GridSAT to solve some of hardest problems using a large collection of computational resources. Finally, we present experimental results for running multiple GridSAT instances to demonstrate the ability of the application to be efficient even in a highly competitive computational environment.

5.1 Splitting strategy

5.1.1 Experimental Setup

In these experiments we determine what fraction of the clause database is shared during the splitting process. We use 31 problems from the SAT competition benchmark. The splitting fraction is varied from 0% to 100% in increments of 20%. It is important to note that the fraction of shared clauses refers to the number of clauses and not to their total size. The total size of the database here is counted as the total number of literals in all clauses in addition to a constant per clause overhead. As an example, sharing the small 50% of the clauses will result in sending less than 50% of the total size of the database. The performance of the solver is based on the total turnaround time of the entire problem set.

For these experiments we used 32 desktop machines where each host used a 2.20 GHz Intel Xeon CPU and 512 MB of RAM. The machines were exclusively used by GridSAT during these experiments.

5.1.2 Results

Table 5.1 shows the results for six experiments. Each experiment was conducted with six different values of the splitting fraction. For each experiment

Shared Ratio(%)	0	20	40	60	80	100
Experiment 1						
Runtime(sec)	31955	37426	31077	30002	36302	34308
% performance loss	6.5	24.7	3.5	(min)	21.0	14.4
Experiment 2						
	35471	29550	26599	31265	29087	33614
% Performance loss	33.4	11.1	(min)	17.5	9.3	30.1
Experiment 3						
Runtime(sec)	30815	35824	30618	30946	28464	44889
% Performance loss	8.3	25.9	7.6	8.7	(min)	57.7
Experiment 4						
Runtime(sec)	41938	38818	32943	31807	32332	33107
% Performance loss	31.9	22.0	3.6	(min)	1.7	4.1
Experiment 5						
Runtime(sec)	30374	30807	29549	29663	32329	29718
% Performance loss	2.8	4.3	(min)	.4	9.4	.6
Experiment 6						
Runtime(sec)	39761	33592	34297	34475	33717	31715
% Performance loss	25.4	5.9	8.1	8.7	6.3	(min)

Table 5.1: Selecting ratio of clause database to send after splitting. The runtimes are in seconds for a benchmark of 31 problems. No clause sharing was allowed in these experiments.

the table shows two rows. The first row displays the total runtime for the entire benchmark. For the second row we select the minimum runtime from the first row, which we denote as *min*, and then we calculate the percent performance loss of the other runtimes relative to it. The percent performance loss for a given runtime rt is calculated as: $\%\ performance\ loss = (rt - min)/min * 100$.

These experiments evaluate different parameter values for sharing clauses during the splitting process. The communication overhead during splitting is very significant since the messages can be 100s of MBs in size. Thus determining what fraction of the database to send can reduce unnecessary network load. According to these experiments, the best ratio to share during splitting varies from 40% to 100%. The variation of the best ratio is expected given that the GridSAT solver is non-deterministic when solving a given problem instance. These experiments show, however, that sharing the learned clauses leads to shorter time to solution most of the time. The best fraction of the clause database to share during splitting is around 60% to 80%. Thus sharing most of clauses is most of the time better than sharing all of it. The reason is that sharing all the database leads the solver to rank the variables in the same order. Therefore it is susceptible to making the same decisions in the future. However, sharing less of the database causes the solver to make different decisions. The overall effect we believe is similar to randomization which is extensively used by most sequential solvers. More over using 80% of the clauses leads to a reduction of more that 50% in the message size. The significant reduction in message size is due to the fact that the fraction of clauses GridSAT is sharing consists in the smallest clauses. The results is significant reduction of the communication overhead while sharing most of the clauses. For the next set of experiments we chose to set the splitting fraction value to 80%.

5.2 Comparing Clause Sharing Strategies

5.2.1 Experimental Setup

In this set of experiments we study the effectiveness of the three different learning methods: the lazy method, the immediate method and the periodic method. These methods use different algorithms to share intermediate clauses as described in section 3.3. The experiments are conducted using a set of 33 benchmark problems used by the different satisfiability competitions [89]. The experiments were conducted on a set of 32 dedicated nodes on a cluster available at the University of California, Santa Barbara (UCSB). The cluster nodes are Pentium IV CPUs with 2.66 GHz frequency and 2 GB of memory. Each experiment uses ten nodes and one of the three methods. The total number of experiments is 297 = 33

Method	**Lazy**	**Immediate**	**Periodic**
Maximum size of shared clause = 5			
Total	76776	68620	64675
% Speedup	*(base)*	10.6	15.8
Maximum size of shared clause = 10			
Total	71860	67292	63400
% Speedup	*(base)*	6.4	11.8
Maximum size of shared clause = 15			
Total	69527	67292	63400
% Speedup	*(base)*	3.2	8.8

Table 5.2: GridSAT results comparing all three learning methods with maximal learn clause size equal to 5, 10 and 15.

problems $*$ 3 shared clause sizes $*$ 3 methods. These experiments are grouped into three sets where the maximal size of a shared clause is varied between 5, 10 and 15.

5.2.2 Results

Table 5.2 shows experimental results for using a maximal size of shared clauses of 5, 10 and 15 respectively. The table contains three sections, one for each of size of shared clauses used. Each section shows to total time for each of the three methods and the relative speed-up compared to the lazy method.

From inspecting each of the three experimental sets, we learned that no particular method outperformed the other two methods all the time. Instead each method outperforms the other two methods for a subset of the problems. We use the total runtime of all the problems to compare the efficiency of the methods. Using the total runtime for all problems in a benchmark is the standard method for comparing solvers.

These experiments help us decide which merging strategy we should use for the GridSAT solver. The merging strategy is important in improving the solver's performance. We notice that in each case both the immediate and periodic methods outperform the lazy method. The immediate method outperforms the lazy method by an average of about 7%. The periodic method was the most efficient and showed a speedup of about 12% on average compared to the lazy method. We also notice that the speedup decreased as the size of maximal shared clause increased. These experiments show that using the periodic method gives the best overall performance.

5.3 Comparison to Sequential Solver

5.3.1 Experimental Setup

The purpose of these experiments is to compare the parallel solver GridSAT to the initial version as implemented by the sequential solver zChaff. In order to outperform the sequential solver, GridSAT should only use additional resources when needed. In addition, GridSAT should amortize the overhead of using any additional resources.

In these experiments we used 34 machines from the GrADS testbed and an additional machine (that we could completely instrument) as a master node. The machines were distributed among three sites: two clusters (separated by campus networking) at the University of TN, Knoxville (UTK), two clusters at the University of Illinois, Urbana-Champaign (UIUC) and 8 desktop machines at the University of San Diego (UCSD). The master node was also at UCSD. The machines had varying hardware and software configurations, with one of the UTK clusters having the best hardware configuration. For each zChaff (single machine) test we used a dedicated node from this cluster.

As a set of test applications, we chose a suite of challenge problems used to judge the performance of automatic SAT solvers at the SAT2002 conference [86]. These benchmarks are used to rate all competing solvers. They include industrial and hand-made or randomly generated problem instances that can be roughly divided into two categories: *solvable* and *challenging* [87]. The solvable category contains problem instances that SAT solvers have been known to solve correctly. They are useful for comparing the speed of competitive solvers since it is likely that each solver in the competition will be able to generate an answer when the competition is held. Alternatively, the challenging problem suite contains problem instances that have yet to be solved by an automatic method or that have only been solved by one or two automatic methods, but are nonetheless interesting to the SAT community. Of these problems, many have solutions that are known through analytical methods, but several are open questions in the field of satisfiability research.

In these experiments the maximum size of learned clauses shared is 10. Learned clauses bigger than 10 are not shared. This size allows for sharing of important clauses that would have maximal effect without increasing significantly the overhead of clause sharing. For the solvable problems we set an overall maximum execution time out to a total of 6000 seconds for GridSAT. That is, if the entire problem is not solved in 6000 seconds, the application gives up and terminates without a definitive answer. For the challenging benchmarks, we double the overall time out to 12000 seconds.

In all of the experiments, we compare GridSAT to zChaff running in dedicated mode on the fastest processor to which we have access with an 18000 second total time out. For the challenging set we used 12000 seconds as the timeout value. Note that in the actual 2002 competition, using faster machines than the fastest we had available, zChaff was only able to complete a few instances from this set using a six-hour (21600 second) time out. Thus we believe that the comparison between the two using the machines in the GrADS testbed offer useful insight into the additional capability provided by GridSAT.

5.3.2 Results

In this section we present experiments comparing GridSAT to the best sequential solver zChaff (according to the SAT 2002 competition). The GridSAT and zChaff solver were run on a set of files that are grouped into three categories. The first category represents the those problems which were solved by both GridSAT and zChaff. The second category represents those which were solved by GridSAT only but zChaff was not able to solve. The final category represents those problem which were left unsolved since neither GridSAT nor zChaff could solve them. In fact these problems were not solved by any other solver according to the SAT 2002 competition.

Now we describe the contents of each of the tables 5.3, 5.4 and 5.5. The second column contains the solution to the instance: satisfiable(SAT), unsatisfiable(UNSAT), or unknown. We have marked those problem instances that were previously open satisfiability problems with an asterisk (*). If a problem was

File name	UNSAT/ SAT/*	zChaff (sec)	GridSAT (sec)	Speed -Up	Max clients
6pipe	UNSAT	6322	4877	1.23	34
avg-checker-5-34	UNSAT	1222	1107	1.10	9
bart15	SAT	5507	673	8.18	34
cache_05	SAT	1730	1565	1.11	34
cnt09	SAT	3651	1610	2.27	12
dp12s12	SAT	10587	532	19.90	8
homer11	UNSAT	2545	1794	1.42	10
homer12	UNSAT	14250	4400	3.24	33
ip38	UNSAT	4794	1278	3.75	11
rand_net50-60-5	UNSAT	16242	1725	9.42	20
vda_gr_rcs_w8	SAT	1427	681	2.10	15
w08_14	SAT	14449	1906	7.58	34
w10_75	SAT	506	252	2.01	2
Urquhart-s3-b1	UNSAT	529	526	1.01	4
ezfact48_5	UNSAT	127	196	0.65	1
glassy-sat-sel_N210_n	SAT	7	68	0.10	1
grid_10_20	UNSAT	967	3165	0.31	12
hanoi5	SAT	2961	1852	1.60	33
hanoi6_fast	SAT	1116	831	1.34	4
lisa20_1_a	SAT	181	243	0.75	2
lisa21_3_a	SAT	1792	337	5.32	4
pyhala-braun-sat-30-4-02	SAT	18	84	0.21	1
qg2-8	SAT	180	224	0.80	2

(*): problem solution is unknown

Table 5.3: Problem solved by both zChaff and GridSAT from the SAT2002 Benchmark Results usning the GrADS testbed. GridSAT shows a significant speedup for the majority of the problems.

File name	SAT/UNSAT/ UNKNOWN	zChaff (sec)	GridSAT (sec)	Max clients
7pipe_bug	SAT	TIME_OUT	5058	34
dp10u09	UNSAT	TIME_OUT	2566	26
rand_net40-60-10	UNSAT	TIME_OUT	1690	30
f2clk_40	UNSAT(*)	TIME_OUT	3304	23
Mat26	UNSAT	MEM_OUT	1886	21
7pipe	UNSAT	MEM_OUT	6673	34
comb2	UNSAT(*)	MEM_OUT	9951	34
pyhala-braun-unsat-40-4-01	UNSAT	MEM_OUT	2425	34
pyhala-braun-unsat-40-4-02	UNSAT	MEM_OUT	2564	34
w08_15	SAT(*)	MEM_OUT	3141	34

(*): problem solution previously unknown

Table 5.4: Problems from the SAT2002 Benchmark which were solved by GridSAT only using the GrADS testbed

File name	UNSAT/ SAT/*	zChaff (sec)	GridSAT (sec)	Max clients
comb1	*	TIME_OUT	TIME_OUT	34
par32-1-c	SAT	TIME_OUT	TIME_OUT	34
rand_net70-25-5	UNSAT	TIME_OUT	TIME_OUT	34
sha1	SAT	TIME_OUT	TIME_OUT	34
3bitadd_31	UNSAT	TIME_OUT	TIME_OUT	34
cnt10	SAT	TIME_OUT	TIME_OUT	34
glassybp-v399-s499089820	SAT	TIME_OUT	TIME_OUT	34
hgen3-v300-s1766565160	*	TIME_OUT	TIME_OUT	34
hanoi6	SAT	TIME_OUT	TIME_OUT	34

(*): problem solution is unknown

Table 5.5: Remaining unsolved problems by GridSAT from the SAT2002 Benchmark Results using the GrADS testbed

originally unknown and was later solved by a solver, then we still keep it marked with an asterisk for completeness. The last column shows the maximum number of active clients during the execution of an instance. For all instances this number starts at one and varies during the run. The maximum it could reach is 34, the number of hosts in the testbed, but the scheduler may choose to use only a subset. This column records the maximum that the scheduler chose during each particular run. When a problem is solved the number of active clients collapses to zero. Speedup is measured as the ratio of the fastest sequential execution time of zChaff (on the fastest, dedicated machine) to the time recorded by GridSAT.

Table 5.3 represents the set of instances which were solved by both zChaff and GridSAT (taken from both the solvable and challenging categories of the SAT2002 benchmark suite since zChaff was able to solve some of the latter). On the small instances (ones that complete in less than 300 seconds) where communication costs are significant we notice that zChaff running on a single machine outperforms GridSAT. The slowdown however is not very significant because the actual time is short. For instances with long running times GridSAT shows a wide range of speed-ups ranging from almost none to almost 20 for *dp12s12*. Because GridSAT was using more machines it was capable of covering much more of the search space even when the run times were comparable. In only one relatively long running instance, *grid_10_20*, did GridSAT show a slowdown. The maximum number of active clients for the entire problem only reached a maximum of twelve during its execution. With this little sharing, parallelism did not seem to improve performance. This particular problem comes from a non-realizable circuit design illustrating the data-dependent nature of SAT solver performance results.

Table 5.4 represent those SAT instances that GridSAT was able to solve while zChaff either timed-out or ran out of memory. In addition, only three out of the ten problems in this category were solved by another solver during the SAT2002 competition [88]. Note that zChaff was crowned the overall winner because of its cumulative performance across benchmarks. Individual instances may have been better solved by particular solvers, but because the competition attempts to identify the best general method, aggregate time is used, and zChaff is the best on aggregate.

Table 5.5 shows the remaining seven instances what have only been solved by GridSAT to the best of our knowledge. Three of the solved instances were part of the challenging benchmark for which results were originally unknown constituting new domain science in the field of satisfiability. The other four had known analytical answers, but no automatic generalized solver had been able to correctly generate them indicating the additional solution power that a Grid implementation brings to the field.

These results show that GridSAT provides a speedup compared to existing sequential solvers. This speed up is not linear with respect to the number of resources used because the DPLL algorithm used to solve SAT instances is a branch-and-bound algorithm. In such search based algorithms the time to solution is not always proportionally related to the number of times the search space is divided. For example, dividing the search space in half may not cause a two-fold speedup in time to solution. In fact, the two sub-problems may have very different times to solution. Actually there is no theoretical guarantee that dividing the search space will result in speedup because of all the heuristics involved. In practice, however, partitioning the search space causes performance improvements most of the time. The contribution of GridSAT is not only to provide speedup over sequential solvers but also to enable the solution of problems that were previously unsolved as shown by the next set of experiments.

The aim of these experiments was to show if GridSAT would realize better performance compared the zChaff. The results in these tables show that GridSAT was able to solve those problems zChaff could solve faster. In addition, GridSAT was able to solve problems that zChaff was not able to solve. In fact these problems were not solved by any other solver. There are, however, those problems that were left unsolved by GridSAT as well as other solvers.

We also tried to compare GridSAT to other parallel solvers. The only other parallel solver we had access to was ParaSatz [58]. ParaSatz would timeout on all instances we used from the benchmark. The reason is that ParaSatz does not use many of the new techniques which distinguish modern solvers.

5.4 Solving "Hard" Satisfiability Problems Using GridSAT

5.4.1 Experimental Setup

Since GridSAT is a true grid application, (robust, portable, heterogeneous, pervasive, etc. [43]), we ran a set of experiments to show that GridSAT can run for extended periods of time robustly using a wide variety of resources and also solve previously unsolved hard satisfiability instances.

In these experiments we simultaneously use computational resources that belong to collections of individual machines, small size research clusters and supercomputing scale clusters. The computational resources we use are composed from four main sources:(1) 40 machines from the GrADS [51] testbed located at University of Tennessee, Knoxville (UTK), University of California, San Diego (UCSD) and UCSB, (2) Blue Horizon at San Diego Supercomputing Center (SDSC), (3) TeraGrid site at SDSC, (4) TeraGrid site at National Center for Supercomputing Applications (NCSA) and (5) DataStar at SDSC. The TeraGrid [105] project is a multi-site national scale project which is aimed at building the worlds largest distributed infrastructure for open scientific research.

During our experiments, none of the resources we used were dedicated to our use. As such, other applications shared the computational resources with our application. It is, in fact, difficult to determine the degree of sharing that might have occurred across all of the available machines after the fact. In batch controlled system such as Blue Horizon, Data Star and the TeraGrid, the queue wait time incurred is highly variable because of jobs submitted by other users.

Thus, if it were possible to dedicate all of the VGrADS resources to GridSAT, we believe that the results would be better. As they are, they represent what is currently possible using non-dedicated Grids in a real-world compute setting.

These experiments also use a more diverse set of resources for longer periods of time (up to a month in duration) and multiple job requests. We chose a set of challenge problems from both [88] benchmarks. These benchmarks are used to judge and compare the performance of automatic SAT solvers at the annual SAT conference. All the problems in the benchmarks are shuffled to ensure that submitted benchmarks are not biased in favor or against any solver. These benchmarks are used to rate all competing solvers. They include industrial and hand-made or randomly generated problem instances that can be roughly divided into two categories: *solvable* and *challenging*. The solvable category contains problem instances that some SAT solvers have solved correctly. They are used for comparing the speed of competing solvers. Alternatively, the challenging problem suite contains problem instances that have yet to be solved by an automatic method or which have only been solved by one or two automatic methods, but are nonetheless interesting to the SAT community. Some of these problems have known solutions that are known through analytical methods (i.e. the problem has a known solution by construction), but several of these problems are open questions in the field of satisfiability research.

In these experiments, we only chose problems from the challenging set. These problems were deemed hard by all participating solvers in both the 2002 and 2003 SAT competitions. We investigate seven previously unsolved problems where three instances are from the SAT 2003 benchmark category, and four are instances from the SAT 2002 benchmark category, all of which we have not been able to solve using previous versions of GridSAT.

This group of problems represent a variety of fields where problems are reduced to instances of satisfiability and solvers are used to determine the solutions. The problems contain a pair of problems in FPGA routing and model checking. These two disciplines benefit heavily from efficient SAT solvers. The remaining problems are of theoretical nature. In addition, we set the absolute minimum size of shared clauses to two and absolute maximum to 15. This range allows for sharing clauses which would help prune the search space without significant communication overhead.

Unlike previous experiments there was no timeout value set for the maximum execution time. Every problem was run using different job description for the batch systems. Jobs on the different batch queues were manually re-launched at random intervals. Job re-submission could have been automated but we wanted more control over rationing our limited compute budgets to specific experiments based on their perceived progress. Experiments where GridSAT was making progress were allotted bigger jobs with longer durations and more nodes. The progress of the solver was judged by inspecting how often the checkpoints were updated. We can also inspect the internal state of a particular solver using some of the tools we developed. The VGrADS nodes were used during the entire duration of each experiment unless the hosts experienced failures.

5.4.2 Results

The experimental results are summarized in Table 5.6. The first column contains the problem file name. The second column indicates the field from which this problem instance in obtained. The third column contains the solution to the instance: satisfiable (SAT), unsatisfiable (UNSAT), or unknown. We have marked those problem instances which were previously open satisfiability problems with an asterisk (*). If a problem was originally unknown when the benchmark was created and was later solved by one of the SAT solvers, then we still keep it marked with an asterisk for completeness. The fourth column represents the total wall-clock time that the problem was tried. Finally, the fifth and last column represents the solution obtained by GridSAT which is represented by SAT, UNSAT or (-) if we terminated the experiment before GridSAT found an answer. Some of these problems were terminated because it did not seem that GRidSAT was making enough progress to warrant the use of valuable supercomputing resources. Note that each problems can be continued later using its last checkpoint.

Table 5.6 shows that GridSAT was able to solve three problems all of which were not previously solved. Two of the problems were found unsatisfiable and they are both from the field of FPGA routing. The first problem *k2fix-gr-rcs-w8.cnf* was solved using the VGrADS testbed only. Batch jobs which were submitted for this experiment were canceled when the problem was solved. On the other hand the second problem *k2fix-gr-rcs-w9.cnf* took much longer to solve, it took more than two weeks. Table 5.7 gives a more detailed description of the resources used during this experiment. For each job a number of GridSAT solver components were launched as indicated in the last column of table 5.7. In table 5.8 a break down of the CPU-hours used on each resource are tabulated. Note that the VGrADS testbed machines were able to deliver a sizable amount of compute power because they were available in a shared mode for the duration of the experiment.

The last problem *cnt10* was also solved using the VGrADS testbed only under similar circumstances to *k2fix-gr-rcs-w8*. We previously tried solving this problem in [28] using the same testbed for four days in addition to Blue Horizon for 12 hours but were not successful. We believe the improvements made to the solver and especially the new clause sharing method have helped achieve this result.

In order to illustrate further GridSAT's success in using all the above variety of resources mentioned earlier we present a section of a run using instance *hanoi6*.

This problem is a SAT representation of the *Hanoi Towers* problem using six disks. A six day snapshot from a 23 day run is shown in figure 5.1 using logarithmic scale. The figure shows several jobs from Blue Horizon, Data Star and TeraGrid sites participating in the execution. This figure shows that GridSAT was able to make use of the available resource when some of their nodes became available and then continued to run after the nodes were taken away to serve other users. GridSAT processes continue to run on the batch controlled resources until the scheduler decides to terminate them. This abrupt termination has no effect on the application which deals with these events as (scheduled) resource failures. GridSAT was able to manage up to 350 processes running on different resources as show in this figure.

Application Efficiency

One of the concerns about distributed applications in general is resource efficiency. The application should be able to make the utmost use of the computational power of the resources in spite of communication and synchronization overhead. The application scheduler should ensure that resources exclusively assigned to the application are always doing useful work and are not idle. In this section we discuss the efficiency of the GridSAT solvers by tracking their CPU usage.

The satisfiability solver performs mostly integer, branching and load-store operations. The number of floating point operations is very low (less than .1 FLOPS). We present in figure 5.2 an estimate of the total number of instructions per second during the same six day period. Since instrumenting GridSAT can cause significant slow down, we conducted some benchmarking on some machines at UTK to determine the average efficiency of the solver. Since the solver code is mostly sequential, we assume that at the maximum only one instruction per cycle can be finished by the processor. The determined efficiency is 70%. We estimated that other hardware and OS combinations will exhibit equal efficiencies. The number of operations provided by a resource is estimated to be the product of its peak performance and the estimated efficiency. The total number of instructions in figure 5.2 is the sum of operations of all active resources. We notice that the VGrADS testbed is able to deliver about 20 *Billion instructions per second* (IPS). In the middle of the graph, there is a batch job from Blue Horizon which failed suddenly while joining the GridSAT execution. This might have happened because the Blue Horizon machine became unavailable for scheduled maintenance. The total number of IPS was multiplied by more than five times when some batch jobs became active. It reached up to 110 $Billion\ IPS$.

Another measure of performance, is how much of the batch job maximum computational power is actually used by GridSAT processes. Most other parallel jobs run on all the processes from start to finish with little overhead. In this case, batch jobs are efficiently used. In the case of GridSAT, however, there are two main sources of inefficiency. First, some jobs might wait ideally at the start. Batch jobs usually include a large number of processes. Some of these processes have to wait until a sufficient number of splits occur to generate new sub-problems for all the newly created solvers. Second, some batch processes may contain idle solvers for a period of time after they solve the previously assigned sub-problem. The solver in this case, waits until it is assigned a new sub-problem by the master. For the first job in figure 5.1, which is a large 100-node job, the efficiency is 98.9%. Thus GridSAT was able to use batch jobs efficiently. The main reason is that batch jobs usually wait in the batch queue for a long time before executing. Thus by the time the job is executed, GridSAT was unable to solve the problem because it is hard. This means that batch jobs are only used when the problem is in deed hard. It is possible that for certain problems, the efficiency of batch jobs might

File name	SAT/UNSAT/*	Time	GridSAT Result
3bitadd-31(T)	UNSAT	8 days	-
k2fix-gr-rcs-w8(F)	*	83261 sec (23 hours)	UNSAT
k2fix-gr-rcs-w9(F)	*	14 days and 8 hours	UNSAT
cnt10(F)	SAT	13134 sec (4hours)	SAT
comb1(M)	*	11 days	-
f2clk50(M)	*	9 days	-
hanoi6(T)	SAT	23 days	-

(*): problem solution initially unknown
(T): Theoretical
(F): FPGA Routing
(M): Model Checking

Table 5.6: GridSAT results using VGrADS testbed, Blue Horizon, Data Star and TeraGrid. All these problems were not previously solved by any other solver.

be low. In this case, future versions of GridSAT might monitor the batch job efficiency to determine whether and when a job is to be terminated.

During our experiments, the Blue Horizon super-computer was being decommissioned and DataStar was released as its replacement. GridSAT was able to continue executing through this transition, starting on Blue Horizon and continuing on Datastar. We continued to submit jobs to BlueHorizon but we did not notice when it stoppped responding. The failure of this single (but important) resource which did not affect the already running experiments shows the robustness of GridSAT.

Compute resource	Job count	Job dur.(hr)	Node count	procs /node
BlueHorizon	2	10	100	3
Blue Horizon	1	12	100	3
DataStar	2	10	8	11
TG@SDSC	1	10	40	2
TG@SDSC	1	12	40	2
TG@SDSC	3	10	4	2
TG@SDSC	4	5	4	2
TG@NCSA	3	10	4	2
TG@NCSA	4	5	4	2

in addition to 40 machines from VGrADS testbed for 14 days 7 hours and 44 minutes

Table 5.7: Batch jobs used to solve the k2fixgrrcsw9.cnf instance from SAT 2003 benchmark

Compute resource	node- -hours	CPUs/ node	CPU -hours
BlueHorizon	3200	8	25600
DataStar	160	11	1760
TG@SDSC	1080	2	2160
TG@NCSA	200	2	400
GrADS(*)	13750	1	13750

(*) machines were shared with other users

Table 5.8: CPU-hours per resource used to solve the k2fix-gr-rcs-w9.cnf instance from SAT 2003 benchmark

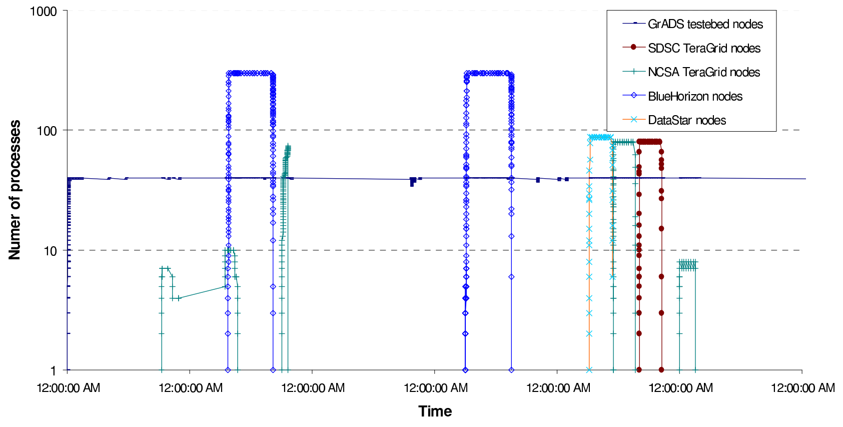


Figure 5.1: A six day snapshot representing GridSAT processor count usage from the different resources in logarithmic scale.

5.5 Running Multiple GridSAT Instances

In this section we present two sets of experiments. The first set of experiments show that GridSAT instances are capable of sharing effectively a common set of resources. The second set of experiments study the effect of executing multiple GridSAT instances using a common resource pool on the turnaround time for solving SAT problems.

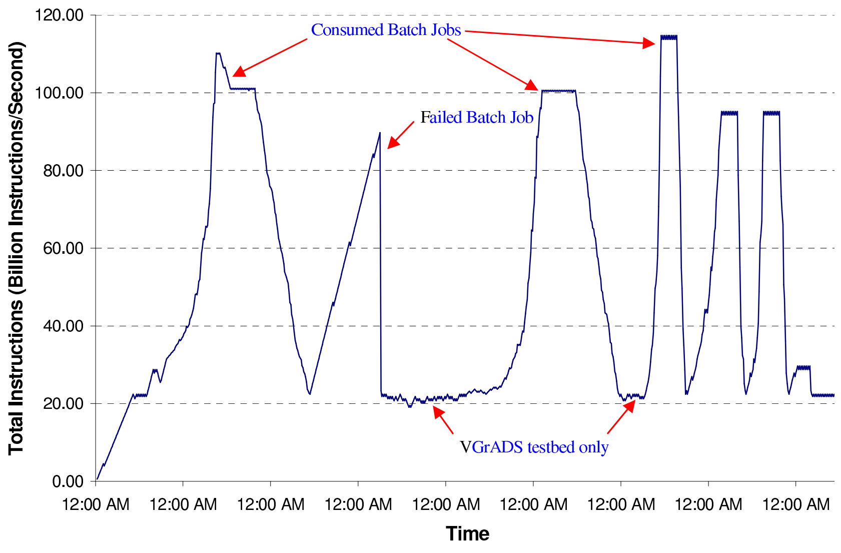


Figure 5.2: Estimation of Instructions per second
usage for all resources during the same six day snapshot shown in figure 5.1.

5.5.1 Interactions between multiple GridSAT instances

Experimental Setup

In this section we study the interaction between two GridSAT instances sharing the same resource pool. We present two experiments to study the effects of stale information. The first experiment uses a version of GridSAT which is oblivious to the effects of stale information. This version shows how the different GridSAT instances suffer from the herding effect. The second experiment, however, uses the techniques described in section 4.5 to mitigate the effects of stale resource information.

Each of the GridSAT instances is started by instantiating the master on the same machine. The master process has very low overhead and does not load the host machine. Thus both master processes do not interfere with each other. Both GridSAT instances are started with the same parameters. The maximum size for shared clauses is set to 10 and the fraction of the clause database shared during splitting is set to .8.

The resource pool is composed of a small cluster of 30 machines. Each of the machines has a 2.66 GHz Intel Xeon CPU with 1 GB of RAM. During these experiments the GridSAT scheduler used the Network Weather Service to monitor resource CPU and memory. NWS sensors were started on all cluster hosts used by GridSAT.

Results

In this section we present two experiments where we compare the behavior of GridSAT resource scheduling before and after we employ measures to reduce the effect of stale resource information. Refer to section 4.5 for details about how GridSAT deals with stale information.

Figures 5.3 and 5.4 show the interaction between two GridSAT instances sharing the same pool of resources. The figure shows how three types of resources change as time progresses. The first resource type is free resource which are not currently running any GridSAT clients. The second type are single resources which are executing a single client from either GridSAT instances. The third resource type are shared resources which is executing two clients one from each GridSAT instance.

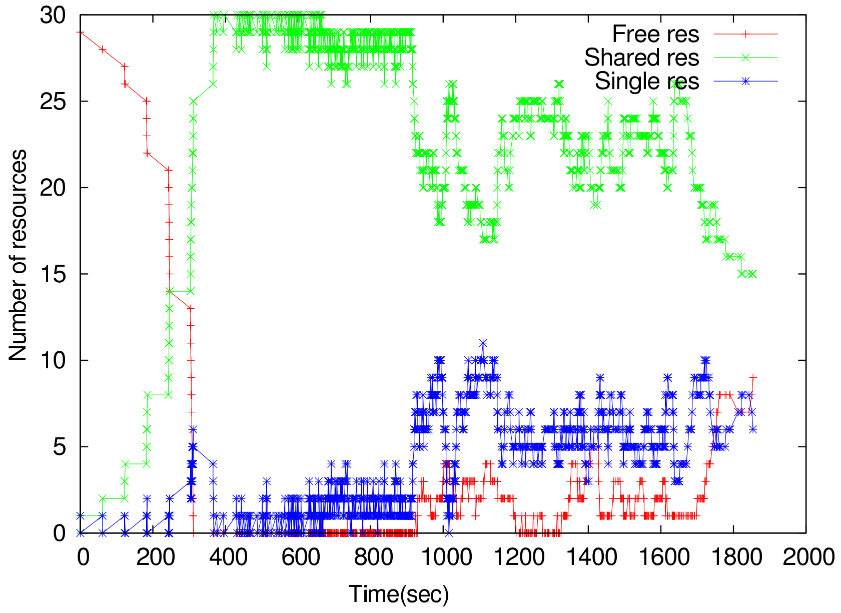


Figure 5.3: Interaction between two GridSAT instances sharing the same resource pool without mitigating the effect of stale information.

In the first experiment both instances were started simultaneously using NWS and allowing the use of stale information. We notice that initially the number of shared resources rises sharply. Actually both GridSAT instances are making the same resource selection. Every increase in singly used resources is followed by a decrease where all those resources become shared. This period lasts from 0 seconds to 380 seconds. This is undesirable ideally both instances should initially be able to use separate resources. We noticed that this is due to the inability of this version of GridSAT instances to detect which resources are really loaded. This same problem persist through out the execution since there continues to exist free resources while the two instances are sharing some of the other resources.

There are two additional phases in figure 5.3. The phase that extends between 380 and 920 seconds is characterized by all resources being in continuous use. In this phase resources are either shared by both GridSAT instances or used by a single instance. This phase shows that both instances are no more making the same decisions. Because of the runtime non-determinism both instances are now making their own decisions and at different times.

In the last phase, which extends from 290 to 1890 seconds, most of the resources are in use. The change in the number of resources used by a single instance and the number of shared resources are almost mirroring each other. This shows that both instances are still making the same resource selection. The changes are not exact mirrors of each other because of the GridSAT non-determinism at runtime. This phenomena could be seen in the second phase but is not as pronounced.

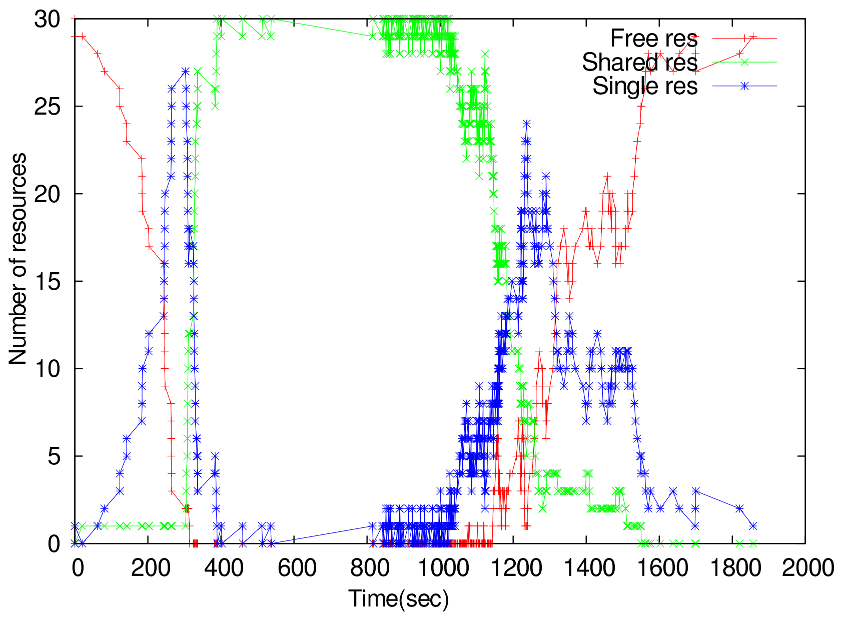


Figure 5.4: Interaction between two GridSAT instances sharing the same resource pool while mitigating the impact of stale information.

In the second experiment GridSAT instances deploy measures described in section 4.5 to mitigate the effects of stale resource information. The measures GridSAT employ a modified resource selection process. Instead of selecting the best resource to launch a new client, the GridSAT scheduler chooses randomly from amongst a set of best resources.

Figure 5.4 shows the interaction between two GridSAT instances launched simultaneously on the same set of resources as the previous experiment. This figure shows three phases similar to figure 5.4. The characteristics and duration of the three phase, however, is very different.

During the first phase from 0 to 400 sec the number of resources used by a single instance increases first. Actually the number of resources used by a single instance reaches 90% of all resources before we see a sharp increase in shared resources. This shows that the instances are choosing different resources to satisfy their resource needs. Since there is no direct coordination between the separate GridSAT instances and because of stale information, there is a possibility that they will sometimes choose a resource that was already in use in spite there are other free resources available. This probability increases as the number of clients being instantiated increases. The phase ends by all resources becoming shared in a very short period. This occurs because the GridSAT instances are splitting at an exponential rate.

The second phase is where all resources are being used. It extends from 400 to 1000 seconds, longer than the previous experiment. In this experiment there is less instances where single resources appear. This is due to non-determinism in the application.

The last phase shows a steady decrease in the number of shared resources and a steady mirrored increase in single resources. In this phase the sub-problems generated by splitting are "easier" and take a short time to solve. In this case more clients terminate than are being instantiated which results in the GridSAT instances having less overall clients. Thus as clients terminate some resources become free as shown at the beginning of this phase. The GridSAT instances in this experiment select these resource for instantiating future clients. Thus the number of single resources increases. This is different from the previous experiment where the GridSAT instances may choose non-free resources leading the a slower decrease in shared resources. Thus this experiment provides better load balance.

The overall effect of reducing the effects of stale information is that the total runtime of the problem being solved in reduced from more than 2000 second to 1800 seconds. In the next section we study the effect of the number of GridSAT instances on the turnaround time of solving SAT instances.

5.5.2 Evaluation of Runtimes for multiple GridSAT instances

Experimental Setup

This set of experiments are designed to evaluate the performance of GridSAT with multiple instances running simultaneously on a common set of hosts. We use 10 hosts that have 3.20 GHz Intel Xeon CPU and 1 GB of RAM. The NWS was used as a service with memory and CPU sensors on each host. All GridSAT instances were assigned the same problem *avg-checker-5-34.cnf*, which belongs to the SAT benchmark. This problem is unsatisfiable.

In these experiments we varied the number of simultaneous instances from 1 to 30. Since the runs are nondeterministic, we performed 10 runs for each of the selected number of instances. For each group of 10 experiments using the same number of instances, we recorded several important metrics.

Results

The results presented in this section include similar plots with the number of GridSAT instances versus the time value in seconds. The time represents three different metrics which we use to evaluate the application performance and trends. The first metric is the time the first problem was solved during each run. The second metric is the time the last problem was solved. The last metric is the average time to solution for all the GridSAT instances which were executed simultaneously. In each of these graph we plot an error bar representing the minimum, maximum and average of each metric within each experiment.

The first graph in figure 5.5 shows the variation of the time the first problem is solved. This time shows the effect of the number of instances on the application startup. As expected the larger the number of instances the longer it takes for the first GridSAT instance to solve the assigned problem. This is due to the larger set of clients competing for resources. So each GridSAT instance gets less clients executing initially and uses less overall computational power. Also the longer error bars indicate that there is higher variation when the number of instances increases. There is a singular application behavior when the number of instances is 10. In this case the variation is small (200 seconds). This behavior is particular to the SAT problem being considered.

The second metric we study in the maximum time taken by any of GridSAT instances to terminate. This metric is shown in figure 5.6. The maximum time indicates the duration when all instances successfully solver the problem and terminate. This metric shows higher variation than the minimum time especially for larger number of application instances. There is also some singular behavior for this parameter when 8 application instances are used. Similar to the previous figure the variation is also smaller (100 seconds) than the general trend would indicate but for a different number od GridSAT instances. We believe that this behavior is particular to the SAT problem being considered.

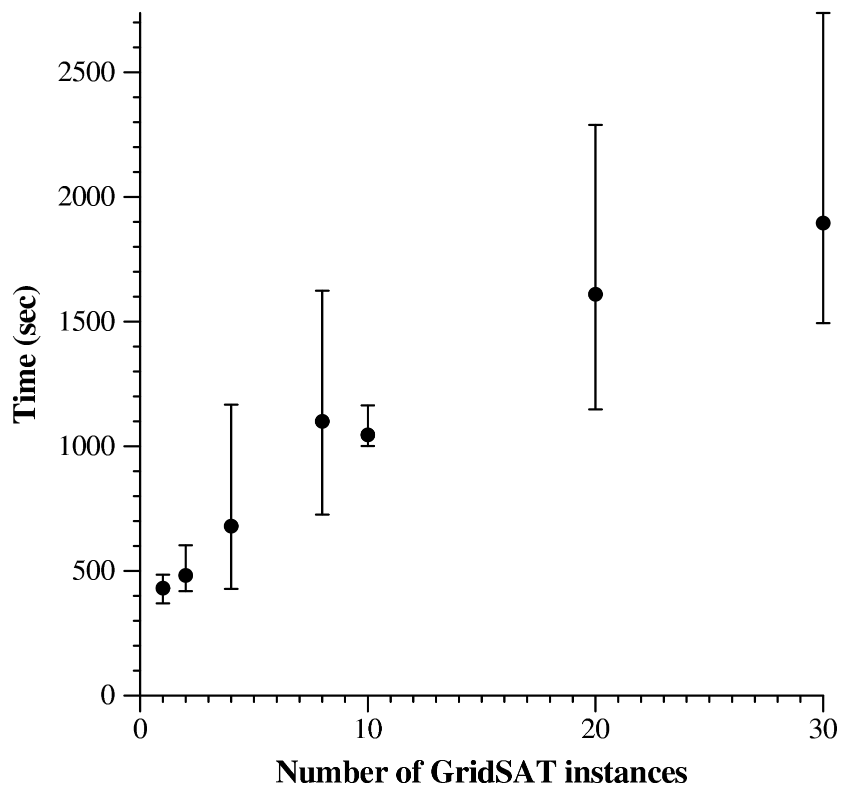


Figure 5.5: Runtimes for the first problem to be solved using a variable number of GridSAT instances running simultaneously.

A related parameter to the maximum time is the average time consumed per instance in a particular experiment. This parameter expresses the average turnaround time of each instance and therefore is probably the most important value for a user. A plot of this parameter is shown in figure 5.7. We notice that this value actually improves as more GridSAT instances share the available hosts. The speedup reaches up to 400% compared to dedicating all hosts to a single

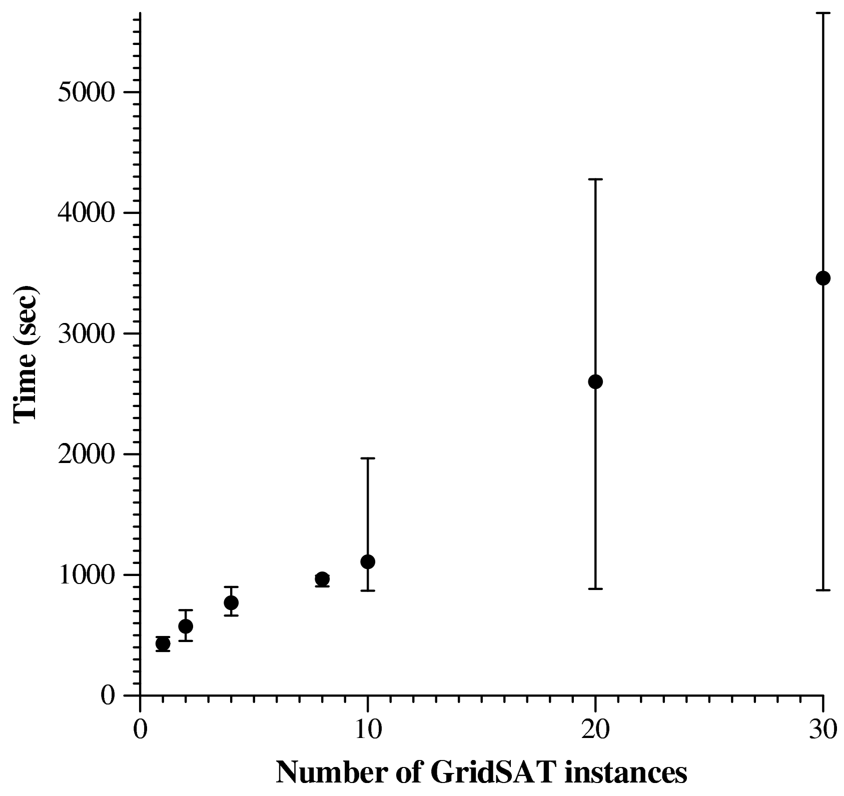


Figure 5.6: Runtimes for the last problem to be solved using a variable number of GridSAT instances running simultaneously.

GridSAT instance. This is an indication that GridSAT is greedy in resource use for this particular problem. The average turnaround time per instance shows little variation and becomes almost constant after the number of application instances reaches 10. This is an indication that the resource set has reached a saturation level as the number of GridSAT instances becomes large. Because of GridSAT's cooperative computing model, those GridSAT instances that cannot instantiate

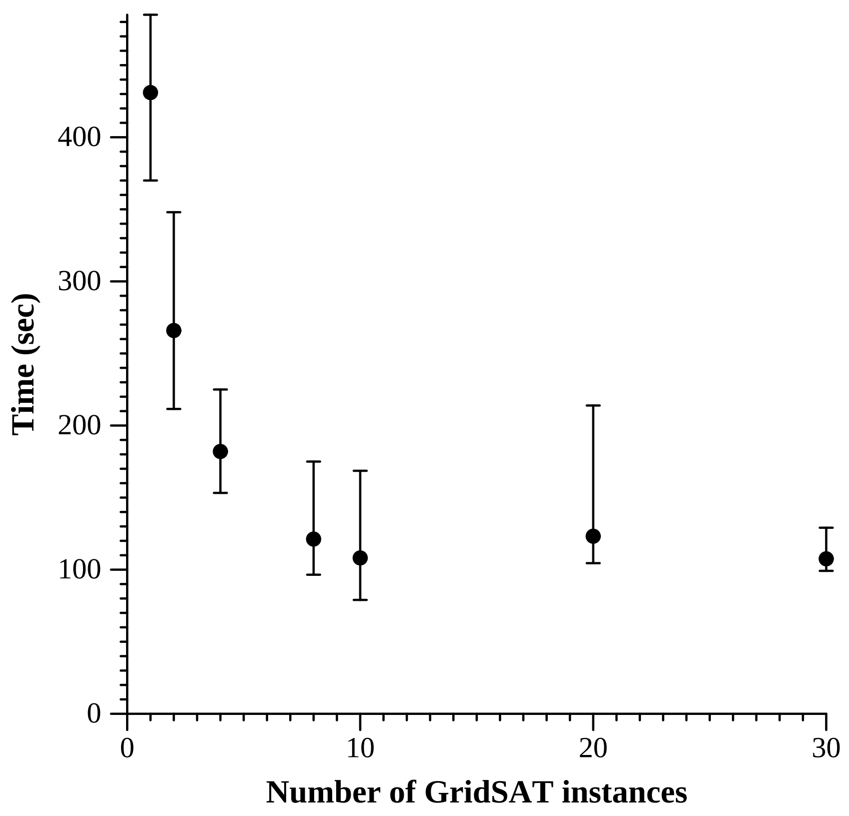


Figure 5.7: Time consumed per instance while simultaneously running multiple GridSAT instances.

new clients wait until some other instance of the application releases some resources. The overall effect is a voluntary queueing system that leaves constant load on the resources. This constant load is defined by the maximum load the resource set can sustain. This behavior enables constant performance even after the number of GridSAT instances becomes very large.

Figure 5.7 shows that GridSAT distributed scheduling is capable of managing a high number of simultaneous instances without reducing the efficient use of resources. In addition, the variability of the time per instance is almost constant and does increase dramatically with the number of application instances until it reaches a saturation level. This further indication that the overall scheduling of resources is stable and does not have negative effects on the application even in such a competitive computational environment.

The last metric is the average runtime for each group of GridSAT instances executed simultaneously. The results are shown in figure 5.8. The trends in this figure are similar to the other figures for the minimum and maximum durations in figures 5.5 and 5.6. The increase of average runtime is linear with respect to the number of instances. Also the variation of this parameter increases as the number of instances increases.

In this section, we have shown that many instances of GridSAT applications can share a common set of resources in a cooperative manner and also improve performance. When the resources are saturated the different GridSAT instances self regulate and wait until less loaded resources become available. In addition, the GridSAT application was able to deploy techniques to successfully reduce the

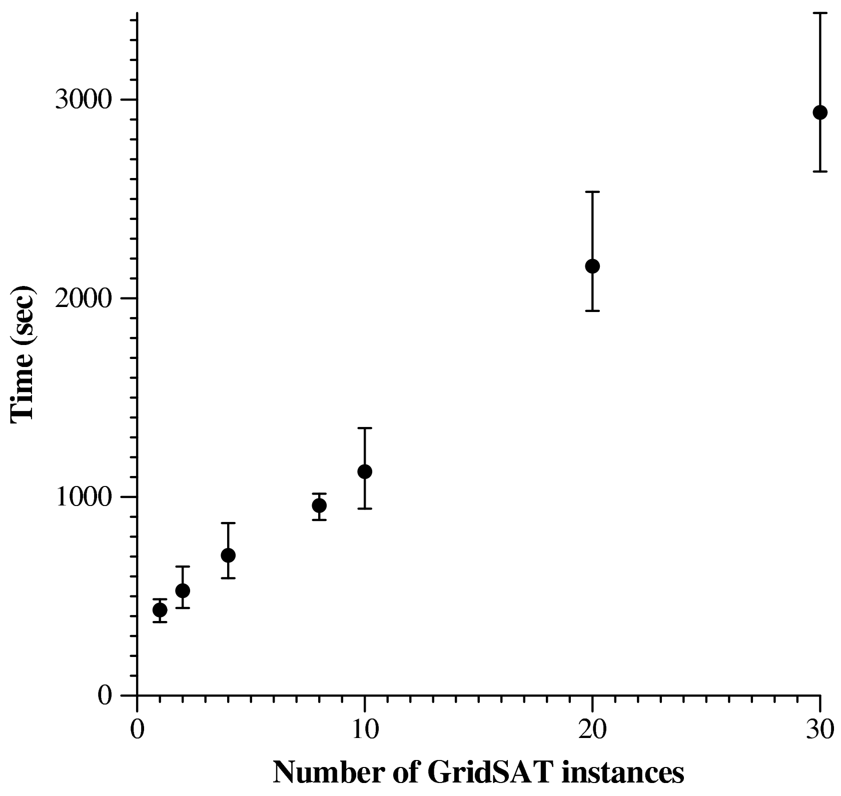


Figure 5.8: Average runtime for all the problems using a variable number of GridSAT instances running simultaneously.

effects of stale information namely. All of this was accomplished in a distributed fashion without direct or global synchronization.

Chapter 6

GridSAT Portal Design and Implementation

GridSAT is a powerful solver and we would like to make this new solving power available to users. The GridSAT system could be made available to users for download and personal deployment. However, two problems exist with this approach. First, the GridSAT system is complex and therefore is hard and time consuming to deploy. Most importantly, not all users are willing to invest in such a deployment. Some users might not have the needed expertise. Second, to solve challenging satisfiability problems from many domains requires using a large set of resources. Most potential users do not have access to such large resources. Moreover, the feedback we have obtained from the community is that users want to simply submit their problems and then get a response with minimal effort. This approach is attractive to many of users especially those with minimal computer science training.

Many complex grid applications face the same challenge as GridSAT because they also need to be made easily accessible to users. Grid applications are usually complex because they have to run in a hostile environment and coordinate the use of numerous resources. This complexity is at odds with the promise of ease of use that the grid computing vision advocates and users unfamiliar with such applications want. One way grid users are presented with an easy to use interface to complex grid applications is through grid portals [12, 4, 7, 2, 9, 1]. The goal of a portal is to hide the complexity at the application and resource management level from users. User effort is minimized by transparently executing the desired application with minimal user input. More advanced users may be given optional access to additional application features. Portals can also improve user efficiency by automating the generation and execution of large sets of jobs. Good examples of previously successful application portal efforts include the Cactus [12] portal for relativistic physics and the Lattice [4] portal specializing in high energy physics.

In order to make this solving power available to interested users, we have developed a GridSAT portal `http://orca.cs.ucsb.edu/sat_portal`. The portal provides a simple and public interface to scientists and other users. In fact, anyone with access to a web browser can submit their specific problem instances. Users are presented with a simple interface and they are prompted to enter a few parameters. Most importantly, the user is insulated from the complex interaction with the GridSAT application and the large number of sophisticated resources.

Developing the GridSAT portal is not merely replacing command line arguments with a web interface. There are many challenges associated with deploying the GridSAT application under the constraints of the portal environment. There are two main categories for these challenges. The first category relates to the portal development, but the second one impacts the application itself. The GridSAT portal shares these challenges with most other portals, but is unique in some respects.

Developing a portal requires the challenging task of using a single user account from the resource's perspective in order to manage and provide access to several portal users. The reason is that as provider of the resources (or allocations on these resources) it is administratively impractical for the portal maintainer to obtain an account for each portal user. Thus, the preferred approach is to give all portal users a unique portal identity but require them to share the same resource user account. It is the responsibility of the portal to manage the different user jobs and provide an accurate accounting of the resource usage for each individual user. The portal would also make sure that the available resources (i.e. disk space, allocation quota) are not exhausted and that the application can tolerate such scenarios.

In addition, most other portals are responsible for complex job of managing the grid resources. This is not the case for the GridSAT portal because the GridSAT application assumes the task of interacting with the computational grid. The application is responsible for automatically selecting the resources and dynamically scheduling the parallel components in order to provide the best performance. This has simplified both the design and implementation of the portal. The portal's role is a simple one; it consists of initiating the application, updating the progress status and collecting the final result.

Furthermore, providing a simple user interface is a challenge from the application's perspective. Such a simple interface requires hiding all the complex parameters and configuration details from the users. For this purpose, the application has to be provided with default parameters when possible. In other cases, it is best to determine such parameters dynamically according to some heuristics. This requires more effort as the heuristics are developed, evaluated and integrated into the application. This approach makes it more attractive to users and also allows the portal to be used as part of a larger work-flow.

In this chapter we first present the portal design and user environment. Then we present some challenges associated with the implementation of the portal. Finally, we introduce the scheduling strategy used for the execution of portal jobs.

6.1 Portal Design

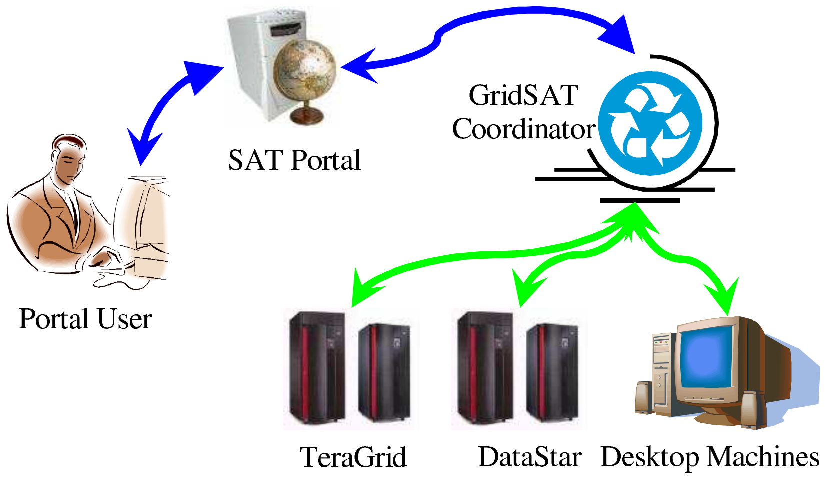


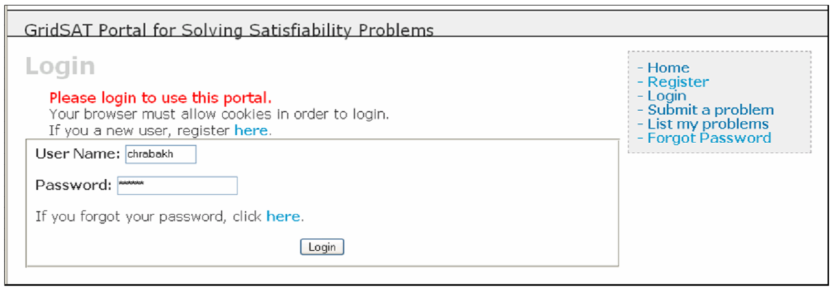
Figure 6.1: Portal design overview: The portal user submits problems through the portal. For each problem a GridSAT instances is launched on the available resources. Feedback is provided to the user through the portal interface.

The general view of the GridSAT portal design is shown in figure 6.1. The portal user can access the portal through a web interface. When the user submits a problem through the SAT portal, the portal launches an instance of the GridSAT solver. During this step, the portal starts the GridSAT coordinator component locally or on a trusted host. The coordinator is started on a trusted host to guard against its abnormal termination. More importantly, if the resource on which the coordinator becomes inaccessible, so then will all the checkpoints saved by the coordinator. If the GridSAT coordinator fails, it can be re-instantiated on another host if necessary. If previous checkpoints are available, the coordinator can use them for recovery. Therefore, it is not necessary to use multiple coordinators simultaneously. It suffices to save multiple replicas of the checkpoints on other resources. The replicas may be encrypted for security reasons. Any of the replicas can be later used to restart the coordinator in case of failure. Client processes can be launched on any available resources because the recovery cost is small. The failure of the coordinator, however, is expensive. Thus this process is not instantiated on shared resources because they are more prone to failure.

Since the master process is long lived, the probability that some of the resources used by GridSAT will fail becomes higher. In fact, in GridSAT we assume that any remote resource may fail at any moment. This may happen because of the resource's own failure or because the resource becomes unreachable through the network. According to our experience, all resources even those which are professionally maintained can become unresponsive from the application's perspective. Those resources that do not experience hardware and software failures usually have scheduled routine preventive maintenance periods or a combination of software and hardware upgrades. From the point of view of the application these are "scheduled" or "anticipated" failures. Thus the GridSAT portal starts the coordinator process locally. It is the responsibility of this process to ensure the continual execution of the application in-spite of resource failures or performance degradations.

6.2 User Environment

In this section, we present the interface to which the user is exposed. For each of the views, we present its function and the significance of each of its components.


GridSAT Portal for Solving Satisfiability Problems

Login

Please login to use this portal.
Your browser must allow cookies in order to login.
If you a new user, register here.

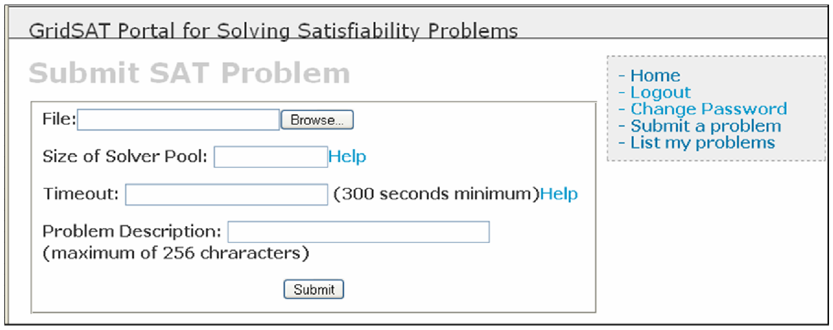
User Name: chrabakh

Password: ******

If you forgot your password, click here.

Figure 6.2: Portal login form
used by users to authenticate to the GridSAT portal.

The portal allows users to create their private accounts where they can enter contact information, username and password. After registering, the user can then securely login to use his account as shown in figure 6.2. Since the portal is for general use, the username and password authentication is sufficient. More sophisticated security systems can be deployed later.


GridSAT Portal for Solving Satisfiability Problems

Submit SAT Problem

File: Browse...

Size of Solver Pool: Help

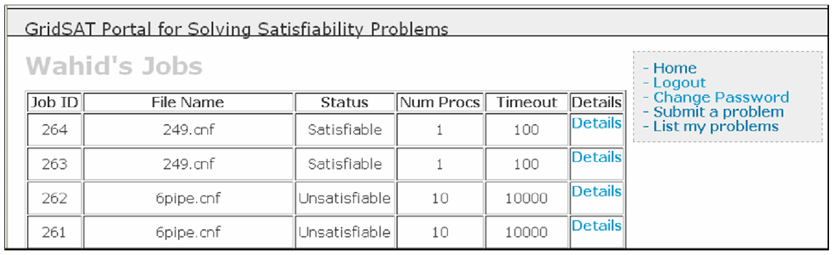
Timeout: (300 seconds minimum) Help

Problem Description:
(maximum of 256 chraracters)

Submit

Figure 6.3: Portal submission form
used by GridSAT portal to submit satisfiability problems.

After logging-in a user can submit satisfiability problems using the form shown in figure 6.3. The user submits problems as files in the standard CNF format. A set of test problems of variable sizes are available for download and can be submitted to the portal at `http://orca.cs.ucsb.edu/sat_portal/test_problems.htm`. The user also specifies the maximum number of processors to use and the duration. The GridSAT application can take more parameters to control the rate of sharing intermediate results and other aspects of the scheduling procedure. The portal hides all these details because most users cannot determine which values to use for these parameters. Instead, the GridSAT application uses heuristics to assign values to these parameters.

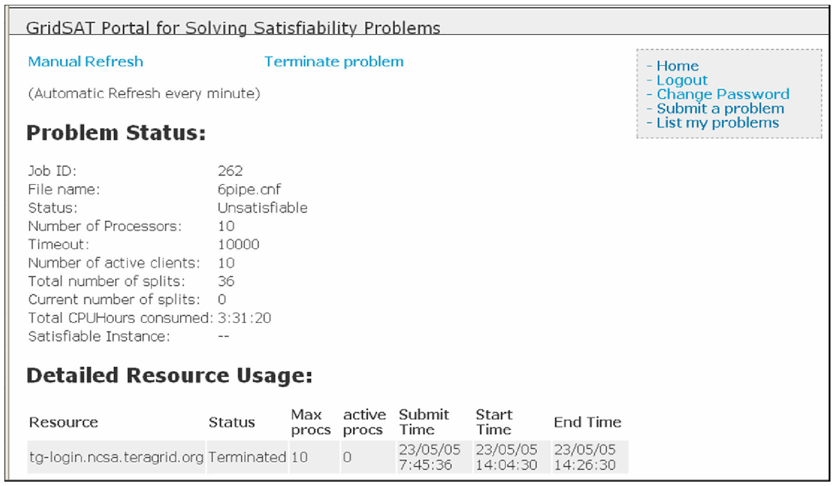

GridSAT Portal for Solving Satisfiability Problems

Wahid's Jobs

Job ID	File Name	Status	Num Procs	Timeout	Details
264	249.cnf	Satisfiable	1	100	Details
263	249.cnf	Satisfiable	1	100	Details
262	6pipe.cnf	Unsatisfiable	10	10000	Details
261	6pipe.cnf	Unsatisfiable	10	10000	Details

Figure 6.4: Portal list view
which shows the user a history of his submissions to the portal. The user can select a detailed view of each problem using this page.

Currently, the GridSAT portal can use many TeraGrid [106] sites located at the National Center for Supercomputing Applications (NCSA), the San Diego Supercomputing Center (SDSC) and DataStar [34] also at SDSC. Additional resources can be incorporated by simply installing the application and updating a configuration file. Managing the set of resources is restricted to the portal administrator. Thus, the user is agnostic to which set of resources will be used. Instead the GridSAT application selects the resources automatically. The scheduler uses resources in a round-robin fashion. But each resource is only assigned a new job after the previous allocation has expired. Thus a resource will only have one job request from GridSAT at a time. Each GridSAT instance can have many waiting or executing jobs over the entire resource pool. In the future, we will use other tools to determine which resource is the most likely to allocate the fastest a given job. The user is also provided with detailed information about the resources used by each problem.


GridSAT Portal for Solving Satisfiability Problems

(Automatic Refresh every minute)

Problem Status:

Job ID:	262
File name:	6pipe.cnf
Status:	Unsatisfiable
Number of Processors:	10
Timeout:	10000
Number of active clients:	10
Total number of splits:	36
Current number of splits:	0
Total CPUHours consumed:	3:31:20
Satisfiable Instance:	--

Detailed Resource Usage:

Resource	Status	Max procs	active procs	Submit Time	Start Time	End Time
tg-login.ncsa.teragrid.org	Terminated	10	0	23/05/05 7:45:36	23/05/05 14:04:30	23/05/05 14:26:30

Figure 6.5: Portal detailed view
of a submitted problem. The page shows several progress statistics and current resource usage details.

The user can query and manage his set of submissions while being provided with continuous feedback. For example, the user can query the portal for all the sets of problems which he has previously submitted as shown in figure 6.4. Moreover, the user can view a detailed status for each problem. A sample view of this detailed status is shown in figure 6.5. The status of each problem is continuously updated. The detailed view of a problem shows the CPU*Hours consumed, the number of active clients are running and the number of total splits which occurred during the elapsed execution time. Moreover, the portal displays, at the bottom of the page, a description of all job requests issued by the GridSAT scheduler. Each job entry includes the resource used and the submission, start and end times of each job. The combination of all this information presents the user with the progress rate at which a given problem is being solved. When a problem is solved the satisfiable solution found is displayed. Otherwise the problem is marked as unsatisfiable. In some cases, the problem might timeout because the specified period has expired before a solution can be determined. Using the details page, the user can also cancel a given problem at any time even when it already started running. A user may also choose to delete the problem altogether using the same page. In this case, the problem will not be displayed in the history of his submissions. Also any disk resources used by the problem will be purged. In fact, all files associated with a given file will be deleted after a given time period in order to alleviate disk space consumption.

6.3 Portal Challenges and Solutions

The GridSAT application is designed to run and adapt to the computational grid environment. Hence, the portal is not responsible for managing the resources and interacting with them. The portal's role is to launch the GridSAT coordinator and update all the related user and problem information in a database. Therefore, the portal is made simpler and easier to develop. However, the GridSAT application has to be adapted to running in a portal environment. In the following we will discuss some of the challenges presented by the portal and how they affected the GridSAT application.

For our portal, we have obtained a set of accounts for several national computing centers supported by the National Science Foundation. We have obtained a single account for each of the computing centers. In other cases, portal providers have the ability to create a separate account per user. This option is not always practical because it requires privileged access to the resources. Therefore, all the GridSAT portal users will in effect share the same account on any given resource. Each account has two main components: allocated time and disk space. With respect to allocated time, the portal has to make sure that users can fairly share the resource. This is accomplished by implementing a maximum number of active problems a user can have at any moment. In this manner, a single user cannot starve jobs submitted by others.

Disk space presents a different kind of problem. Most resources assign a specific disk quota for each account. Thus all portal users will share this disk space. In GridSAT, this disk space is used as cache to store problem files. These files can be up to 100 MB in size. Storing these files makes the GridSAT solver more efficient by reducing communication overhead. However, if the number of problems is large, the disk quota may be overwhelmed. In order to efficiently use the disk space, all problem files are deleted when a problem terminates. Also, the GridSAT components will use the disk as cache space only if there is enough space to store the problem file. Thus, GridSAT clients check available disk space every time the problem file is requested by a client. When the disk space available is not sufficient, the client receives the problem file in memory and is used immediately without saving it to disk.

In the next section, we discuss the scheduling policy used by GridSAT in the portal setting. We adopted this policy, in order to simplify and automate scheduling for portal users.

6.4 Budget Based Scheduling

In order to adapt the GridSAT application to the portal setting, a special scheduler has been adopted which requires minimal user input. As shown in figure 6.3, when submitting a problem the user specifies two additional parameters which are the maximal number of processes and a maximal duration for trying to solve the satisfiability problem. The GridSAT scheduler uses only these two parameters to automatically schedule the application on all the available resources.

Since it is not always possible to fulfill exactly the user requests through a single allocation, the GridSAT scheduler uses these two parameters only as guidelines for submitting resource requests. For example, if the user asks for a number of processes greater than the number that can be provided by the resources available, GridSAT uses the maximal number of processes available within the resources instead. Also when a user specifies a small time duration lower than a minimal predetermined value, then all jobs requested will use the minimal time value instead. The portal enforces a minimal time for using a CPU to avoid inefficient resource usage. In general, using a CPU for a very short time period does not allow the solver enough time to make progress in solving the problem it is assigned. Hence, a minimal duration for each job is enforced to avoid wasteful use of resources.

Effectively, the GridSAT scheduler tries to satisfy the user requests within the constraints of the resources available. If it is not possible then the scheduler submits a series of smaller jobs with equivalent total computational budget in CPU*Hours. The scheduler keeps count of how much of the budget has been consumed by the application. The remaining portion of the initial budget is decremented continuously as processes are executing on behalf of the application. When a job terminates, the scheduler uses the remaining budget and resource specific parameters to submit a new job.

Chapter 7

Related Work

The GridSAT application and portal share commonalities with two main research fields. The first area includes satisfiability solvers both sequential and parallel. The second area includes many aspects of computational grid computing such as programming models and scheduling. In this chapter, we present related work in these areas and draw comparisons with GridSAT.

7.1 Satisfiability Research

7.1.1 Sequential Solvers

There has been extensive research efforts focused on the development of efficient satisfiability solvers [70, 48, 53, 19]. Traditionally, problems from practical domains [96] were solved using other tools such as Binary Decision Diagrams

(BDD) [54]. But since satisfiability based solvers have become more efficient they have replaced older tools.

SAT solvers solvers use different techniques and heuristics to explore the entire search space. The most efficient of these solvers use optimizations which permit parts of the search space to be discarded or “pruned” during execution. However, because the general problem is NP-complete, there is no theoretical framework for comparing solvers or evaluating which solver is best suited to a particular problem or problem class. The solvers and the techniques they implement are evaluated based on empirical results by comparing the speed with which they can solve a diverse set of benchmarks and/or the number of complex or “hard” problems they can solve. Thus while the general problem remains theoretically intractable, heuristic-based approaches have yielded SAT solvers that serve as valuable verification tools in many disciplines.

Most modern solvers [70, 48, 53, 19] are sequential and employ heuristic improvements to one of a small set of fundamental search algorithms. Fewer parallel solvers such as [28, 58, 97, 41] exist, and even fewer of those parallel solvers use a heretofore sequential optimization termed *learning*. Learning (discussed in detail in chapter 2) improves solver speed by adding propositions that the algorithm deduces to an internal database that is global to the solver. These additional "learned" propositions improve the efficiency of SAT solvers substantially, but they make the problem of parallelizing and/or distributing a solver daunting. The global clause database must be searched and updated frequently as the algorithm progresses making an efficient large-scale parallel or distributed implementation difficult. As a result, the best known solvers (in terms of speed and solution power) have until recently been sequential.

Since SAT problems can be expressed in a standard format then they can be submitted to any solver. Modern solvers and algorithms, however, are targeted by design to solve faster certain types of SAT instances. This affinity for certain problems derives sometimes form the algorithm adopted by a given solver. For example, implementation of Iwama's [55] algorithm are faster for random instances with comparatively *many* solutions but slower for those instance with *few* solutions. Solvers using simple backtracking, however, are *slow* on problems with *many* solutions and *fast* for instances with *few* solutions.

In some cases experimental results are used to differentiate the performance of SAT solvers with respect to certain categories of SAT instances. According to the SAT competition different solvers perform differently depending on the problem category. Some of the problem categories include industrial benchmarks, randomly generated, satisfiable and unsatisfiable instances. The SAT competition declares winners for each of these categories those solvers which solver the most problems using least time duration.

GridSAT can be abstracted as a set of collaborating sequential solvers. These collaborating solvers are not required to be homogeneous. In fact, GridSAT can incorporate many sequential solvers. This can be accomplished by deploying clients with different sequential solvers at their core. Any existing or new sequential solvers may be modified to cooperate with the rest of the clients in order to enhance performance. As a first step it is possible to deploy these solvers without modification. This is possible my formulating sub-problems as separate satisfiability problems. Such a task can be simply implemented by adding all variables in the first decision level as unit clauses. Therefore GridSAT can take advantage of developments in sequential solver research with little or no extra effort.

7.1.2 Parallel Solvers

Many research efforts exploited parallelism to speedup procedures used in SAT solvers. Generally, two approaches were used. The first approach exploits a software architecture to mediate the cooperation of many CPUs in order to provide more computational power. The second approach studies inefficiencies in the implementation of conventional processors and proposes a new hardware architecture which is better suited for solving SAT problems.

Existing deployments of parallel solvers use limited resources [58, 97, 41, 21]. Actually, most reported experiments deal with problems of short durations (i.e. less than a minute). Therefore, more difficult problems which require larger sets of resources for extended periods of time were not investigated. Using such large collection of resources is a perfect fit for computational grids. The parallel solver, however, has to explicitly adapt to this environment in order to improve performance and enable new results.

GridSAT can also make use of other parallel solvers in a similar fashion to the way it can incorporate sequential SAT solvers. Other parallel solvers can be assigned parts of the search space in the form of a CNF SAT instance just like their sequential counterparts. Parallel solvers, however, represent an additional challenge because of the many resources they can use simultaneously. Some heuristics need to be developed to select the size of resources a given parallel solver is allowed to execute on.

Software Based Solvers

Most parallel solvers [58, 97, 41, 21] use a software approach. Also the majority of these solvers evolved from a sequential solver. PSATO [122], for example, is based on the sequential solver SATO [121]. PSATO is concentrated on solving 3-SAT and open quasi-group problems. An other solver is Parallel SATZ [58] which is the parallel implementation of SATZ [64]. Unlike GridSAT, both solvers only use a set of workstations connected by a fast local area network. This setup results in low communication overhead. PSATO and Parallel Satz do not include clause exchange. PaSAT [97] implements a different algorithm for clause sharing. In addition, PaSAT uses a *global lemma(clause) store* whereas GridSAT shares clauses globally as soon as they are generated.

A different approach is presented by NAGSAT [41]. Instead of search space partitioning, NAGSAT uses *nagging* to enable asynchronous parallel searching. Nagging uses a master node which proceeds as a complete sequential solver. The clients or naggers request a search subtree and apply a problem transformation function. The master incorporates any valuable information returned by the clients. This solver was only applied to a set of randomly generated 3-SAT instances.

MASSAT [119] is representative of incomplete solver. It is based on a distributed multi-agent approach which uses local search method [92, 91]. MASSAT divides the SAT variables in to groups. Each group is in turn assigned to an agent. Each agent *lives* in an environment represented by the local search space. Agents navigate their respective environments based on some reactive rules. These rules define the next move an agent will make in its environment. Moves are evaluated based on heuristic functions which give different weights to clauses. At each iteration, agents communicate with each other to exchange state information. This information includes the variable settings within each agent. Using this information in combination with the specified reactive rules an agent changes the values assigned to its set of variables to move closer to finding a solution. The solver terminates when a solution is encountered or a pre-specified time limit is reached. A solution is the combination of all agent states or positions within their environments.

Another parallel solver is ZetaSAT [21] which targets the zetagrid [112] desktop grid platform. Unlike GridSAT ZetaSAT is not capable of using batch controlled systems which can provide substantially larger computational power compared to desktop machines. Also ZetaSAT does not employ sharing of clauses which permits faster pruning of the search space.

ZetaSAT maintains a pool of sub-problems so that work can be assigned to idle resources for load-balancing. The sub-problems in this pool are pre-generated in anticipation of new and/or idle resources becoming available. In GridSAT, however, clients generate new work only after idle resources become available. This approach ensures that splitting is only performed when new resources are available to help solve the current SAT instance. Also, the splitting process in GridSAT uses peer-to-peer communication bypassing the coordinator (also called master) for large messages. GridSAT also provides newly split clients with a large set of learned clauses which increases overall efficiency.

satinsat

Hardware Based Solvers

Hardware solvers deploy novel arrangements of processing elements to speedup critical procedures of the SAT solvers [98, 99, 77, 33]. These solvers use the same high level DPLL based algorithms used by their software counterparts. Since BCP accounts for a large portion of the runtime, hardware implementations focus on reducing BCP overhead. These solvers implement faster BCP using parallel processing of clauses during this step. The techniques used by these solvers range from connecting multiple chips together to redesigning circuit layout of processor units to target a specific SAT instance. In the following we present two example solvers.

A parallel scheme based on a multiprocessor implementation is presented in MPSAT [125]. The configurable processor core was augmented with new instructions to enhance performance. Like typical hardware solvers data parallelism is used to speed-up execution of common functions in the DPLL algorithm. This solver however, uses standard FPGA circuits to implement the desired optimizations.

Easily-Loaded Variable Implication Solver (ELVIS) [23] uses an instance specific layout of processing elements. In this case the placement and routing of processing elements are designed to provide greater speedup based on a priori SAT instance analysis. ELVIS provides an automatic layout scheme which reduces the overall latency in the final chip design.

Unlike GridSAT and other software based solvers, the hardware approach relies on specialized processing elements which has higher cost and is not easily available to many potential users of SAT solvers.

7.1.3 Related Problems

There are many other problems which can be parallelized in a similar fashion to the satisfiability problem. Most of these problems are formulated as search problems. They are characterized by vast search spaces which can be divided to allow the exploitation of many hosts in parallel.

Some of these problems derive directly from SAT. Some examples include finding SAT solutions with specific criteria such as the ones discussed in section 3.4. Other problems are from the more general class of Constraint Satisfaction Problems (CSPs). For these problems new algorithms need to be developed to enable large scale parallelization.

Also some other problems derive from preprocessing techniques to reduce the search space of SAT instances. One such technique is symmetry breaking [39]. This procedure adds new clauses to the initial SAT problem to remove redundant variable combinations. Finding these clauses involves finding isomorphisms in a graphical representation of the SAT problem. The solutions to the isomorphism problem are time consuming and may be parallelized to improve their performance.

In addition there are more theoretical problems such as Rieman's hypothesis [81] and Ramsey numbers [79]. These problems currently have no theoretical solutions and software can be used to prove or refute certain results. In some cases the software developed is used to help expand knowledge about these problems.

7.2 Computational Grid Computing

There has been extensive research aimed at facilitating the computational grid vision. Some of the problems faced by grid computing are shared by other fields. Such fields include traditional fields such as parallel programming and newer research areas such as peer-to-peer [46, 85, 82, 124] and Web-based computing [93, 63, 108]. Since they share common problems, techniques developed in one field can be applicable to other areas. For example, peer-to-peer systems have developed methods for reliable access to stored data that can be used by computational grid application [49]. This may eventually lead to these fields converging since they are seen to be solving closely related problems. In computational grid computing research efforts can be classified into three different groups:basic services, applications and development environments.

7.2.1 Services

A typical simplified view of a computational grid application is shown in figure 1.1. Since many applications have common needs before deployment, initial research has identified and developed certain basic services. These basic services include communication, resource management, resource monitoring and security.

Communication

There exists many commercial technologies such as CORBA [74], JRMI [24] and DCOM [52]. These technologies [76] provide high level abstractions, platform independence and and code reuse. These features are very important. However, these models hide essential information in order to achieve a simplified view of the computational environment. Information that is essential for grid applications to extract high performance from a given resource. For example, a grid application can tune its code based on the available hardware to achieve better performance. A grid application would also need detailed information about the entire resource pool in order to select the set of resources which would increase efficiency and reduce overhead.

Because a grid application aims to coordinate the use of many distributed resources, efficient and reliable communication between all components of a grid application is very important. There are many alternative communication layers for a grid application. Some layers are part of a more comprehensive set of tools such as in Globus project [42]. Other tools are devoted to the single task of providing a portable communication library such as EveryWare [114].

Resource Discovery

Another important service deemed necessary for grid applications is resource discovery. The purpose of this service is to provide up to date detailed information about the software and hardware characteristics of the entire collection of available resource in a computational grid. The Globus Metacomputing Directory Service (MDS) [31] is such an example. The Globus MDS relies on a set of protocols to query and update resource specific attributes.

Resource Monitoring

Because grid applications run in a dynamic environment, resources do not have static performance characteristics. Therefore, monitoring different aspects of the computational environment such as CPU and memory usage, disc space available and network connectivity. There are many monitoring systems [117, 66, 38] which can be used by an application. These systems are usually deployed independent of the resource monitoring system since they provide dynamic data which changes frequently. In some case, monitoring systems are incorporated into resource management systems.

Execution Systems

An important service is provided by execution systems which allow applications to remotely execute programs on a pool of resources. Such a systems is the Globus Resource Access Manager (GRAM) [32] and Condor [104]. These systems provide remote job instantiation using specific languages. Also these systems also include some security mechanisms to protect users and resource providers.

7.2.2 Applications

Another important research effort in grid computing is to enable applications to execute in a dynamic heterogeneous environment. The applications deployed in computational grids thus far can be divided into two main categories:*embarrassingly parallel applications* and *fixed granularity applications.*

Embarrassingly parallel applications [10, 100, 26] require large computational power but they can be divided into many smaller jobs. These jobs can usually be assigned to a single CPU resource and can be solved in relatively short time periods. Each job executes the same program using a different set of inputs. Also, the individual jobs are independent so that the execution of one job does not depend on results from another one. Therefore, these type of applications adapt well to a computational grid environment. In a typical execution of such applications, all the jobs are generated and then assigned to available resources using a master-worker topology. The most famous of these applications are parameter sweep applications [10, 26]. A parameter sweep application, divides each parameter space of an application into intervals. Each tuple of parameter values produced by the cross product of the parameter sets defines a sub-task. There has been extensive research into grid implementations for such applications as groups of tasks [10, 26].

Since these applications have common features, they can be automated using a development environment. Nimord [10] and APST [26] are two such examples. Nimrod provides a flexible userinterface used to generate a job list according to parameter lists and values. The tasks are automatically scheduled and monitored. Nimrod/G incorporates Globus services and the scheduler is aimed at meeting userspecified deadline using budget constraints. The AppLeS Parameter Sweep Template (APST) is a middleware based on AppLeS [100]. The goal is to minimize the application makespan. The architecture of APST allows for the plugin of other schedulers conforming to the scheduling API. APST consists of the following modules: scheduler, controller, actuator and MetaData Bookeeper. The controller interfaces the scheduler and the client. The actuator controls access to Grid resources. The Bookkeeper is responsible for getting forecasts and saving application generated measurements.

Another set of applications which has been deployed in the context of computational grids are *fixed granularity applications*. These applications involves high performance tightly coupled components. The general execution model in these instances is that each component alternates between small intervals of communication and computation. Usually communication and computation intervals intersperse the execution time and do not overlap. The communication intervals are used to perform synchronization and updates of common data. These synchronizations may be global or pertinent to smaller groups of components depending on the scope of data being shared. Most of these applications are MPI-based [69] scientific applications.

The performance of fixed granularity applications is highly sensitive to heterogeneity of the underlying resources and to network delays. The reason is that these applications are tightly coupled and require frequent communication. Thus, all the components will wait idly if one of them is delayed because of degradation in computational or network performance. As such these applications are not deployed in highly variable environments. Instead the schedulers of these application query the grid environment for sub-grids which are semi-homogenous in computational performance and network connectivity.

The Grid Application Development Software project (GrADS) [18] and its successor the Virtual Grid Application Development Software are two research project which aims at enabling is a comprehensive set of tools and production testbed for developing grid applications. There are many applications [109, 83] which were deployed to validate the GrADS concepts. The scheduling problem in these applications was devoted to selecting amongst a set of available clusters.

These two application categories represent many applications. These two application types, however, represent two extremes of the spectrum of applications. One requires no sharing of information between the sub-tasks, while the other has stringent frequent synchronization. There are many other applications categories which have not yet been investigated in the context of computational grids. For example, a set of applications with intermediate characteristics can also benefit from a computational grid deployment. Such *malleable* applications can adjust the amount of communication and computation based on their cost in a dynamic environment. Further, research is needed to identify and explore new applications which may require new programming models.

7.2.3 Grid Computing Environments and Portals

There is a proliferation of grid computing environments. These environments present simple interfaces to powerful tools and applications that are made available to non-expert users. The user is usually presented with a web portal or a simple command shell.

In current practice, grid computing environments are built using a stack of technologies. At the bottom are the capabilities of the resources which makeup the computational gird. The second layer is composed of the basic tools providing the basic services described above. The Globus tool kit is often used to implement this layer and provide access to grid resources. In the third layer, these basic services are wrapped in commodity technology to make them available to programmers familiar with these technologies. The Java Commodity Grid (CoG) Kit [111] is such an example which wraps Globus capabilities in a familiar java framework. Such technologies are themselves presented as a middleware such JPDK [72]. Finally, a Problem Solving Environment (PSE) targeting a single or a fmaily of application is built based on these middleware packages.

Portals have become the main vehicle for giving common and non-expert users access to sophisticated grid applications. In the next section we discuss in detail portal development tools and the GridSAT portal.

Portals and Portal Development tools

In general, there exist two classes of portals. The first class is user portals which provide an interface for grid infrastructure to users. An example user portal is the NPACI portal [6]. The other class of portals is application portals. These portals provide a simplified, easy-to-use interface to start, stop, manage and monitor complex applications in a specific field. The GridSAT portal belongs to the latter category.

Since user portals provide a set of standard services, some tools such as GPDK [72] have been developed to streamline the development of these portals. Application portals, however, are more specific because each application solves a different problem and therefore provides different services. In spite of this, there have been attempts at building tools for such portals. There are two approaches adopted by tools used in building application portals. In the first approach, a set of small components are provided which can be composed and customized to create the portal. Such projects include GridSphere [3] and JetSpeed [16]. The small components are called portlets. Portlets are usually small software modules aimed at providing a specific functionality. Portlets make portal development easy because developers can reuse existing standard portlets. There are many standard portlets which can be readily used to provide basic functionality common to most portals. For example, there exist standard portlets for user registration and security. Portals developers can also deploy customized versions of existing portlets or develop new ones.

The second approach provides a complete mechanism for building the portal. These tools are only suitable for a certain class of portals which provide a standard set of features which can be easily abstracted. For example, GridSpeed [102] abstracts portals into a mechanism for specifying parameters and a predefined set of templates for task execution. Under this abstraction, the purpose of the portal is to initiate commands on remote systems. This model is useful for many legacy applications or in certain cases where only a binary is available. In both cases, it is difficult to transform the application into a grid application.

The GridSAT portal is different from most existing portals in two main aspects. First, it is the GridSAT application that is responsible for launching and monitoring the tasks on the remote resources. In most other portals, the portal is intimately involved in resource scheduling. Second, for each problem launched by the GridSAT portal the number of tasks, the number of resources and duration of each task is not predefined. This differs from other portals where each problem is assigned a priori a set of resources and the expected execution time is known. The GridSAT scheduler, however, dynamically chooses the next job characteristics (i.e. location, size and duration) depending on the observed resource load and problem behavior from previous jobs.

Therefore, the GridSAT portal cannot be built using the second approach as it does not fit a simple abstraction. On the other hand, portlets can be adapted for this portal. Some of the standard portlets could be used to provide basic feature such as user registration. But new portlets are needed to provide GridSAT specific functionality. Because of the programming effort and time constraints we were not able to use portlets in the current version. Currently, we are using an APACHE [15] web server with PHP [8] and MySQL [5] database to deploy the portal.

Chapter 8

Conclusion and Future Work

In this thesis, we have presented a new computational grid application. The application solves boolean satisfiability problems which have many practical applications in many scientific and engineering fields. This application is GridSAT – a distributed large scale satisfiability solver.

The solver was successful at many levels. First, it was able to solve faster those problems that could previously be solved by existing sequential solvers. Our solver also provides increased solving capability by solving additional problems that were previously left unsolved by other solvers. In addition, the solver was able to simultaneously execute on a large collection of resources for extended time periods in spite of the dynamic nature of the computational environment in terms of performance and composition.

The success of the GridSAT solver comes from two aspects. First the parallelizing approach allowed incremental division of the problem. This results in an approach which decreases communication overhead for both easy and hard satisfiability problems. Also, the ability to share information between the parallel components enabled additional solver efficiency. Second, the implementation and design of the application resulted in the ability to use all available resources. These resource included single desktop machines, small scale clusters and several national super-computers. The ability to tailor the solvers memory use to all these resource was instrumental in making the application components portable.

8.1 Future Work

The work described in this thesis can be expanded further in four main directions. The first direction explores more efficient algorithms specific to satisfiability solvers in both sequential and distributed form. The second direction explores other challenging applications which have similar characteristics to GridSAT. The third area of future research is the investigation of different schedulers. Finally, all the experience learned can be provided in as programming tools to simplify the deployment of future grid applications.

8.1.1 Algorithmic improvements and resource specific implementations

In the future, the application components could be further ported to new resources with different hardware and programming models. For example the BlueGene [107] architecture presents a large number of compute nodes with local memory. Another example, is the Cray MTA [14] and Eldorado [40] machines are shared memory machines with a large number of hardware level threads named *streams*. Algorithmic innovations are needed to make efficient use of these innovative architectures. Once the GridSAT components are ported to these machines they can be seamlessly integrated to the overall application using the existing design.

8.1.2 New applications

There are many other problems which could be deployed in a computational grid environment such as those described in section 7.1.3. Some of these problems belong a large class of Branch-and bound problems. Others, involve large datasets and are compute intensive. once these applications are developed they can be made available using a portal for general use. Also these applications can provide a web-service interface so that they can be automatically included in larger workflows.

8.1.3 New Schedulers

There are further improvements that can be applied to the GridSAT scheduler. The scheduler could make use of resource prediction research [25] to make more efficient resource selection. Such a scheme would allow the application to acquire the most computing power in a given time period.

Another area that can researched further is using different types of schedulers for the GridSAT application. One possible approach is to use economic based schedulers for resource evaluation and selection. In this approach, resources are evaluated based on their cost to the user as a function of a pricing mechanism adopted by a computational grid. Economic based schedulers represent a different challenge because the application has to evaluate resources not only based on their computational merit but also on their value within an economy.

8.1.4 Generalized Tools

The lessons learned from and the programming techniques used in developing a few of these applications can be generalized as a programming model for other applications which share common features. The final tools for programming this class of applications can be expressed as either libraries, programming languages or middleware. As a result, many more applications can be easily deployed on computational grids. Thus, bringing the vision of easy and simple grid computing closer to realization.

Bibliography

[1] GeneGrid Portal. http://www.qub.ac.uk/escience/projects/genegrid/.

[2] GEON Portal. http://www.geongrid.org.

[3] The GridSphere web site. http://www.gridsphere.org.

[4] Lattice Portal. http://lqcd.jlab.org.

[5] MySQL. http://www.mysql.com.

[6] NPACI HotPage. https://hotpage.npaci.edu/.

[7] NRC Grid Portal. http://www.grid.nrc.ca.

[8] PHP. http://www.php.net.

[9] Theoretical Physics Portal. http://www.aus-vo.org.

[10] High performance parametric modeling with nimrod/g: Killer application for the global grid? In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 520, Washington, DC, USA, 2000. IEEE Computer Society.

[11] *Solving the Round Robin Problem Using Propositional Logic*. AAAI Press / The MIT Press, 2000.

[12] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment. *The International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[13] G. Allen, T. Goodale, T. Radke, M. Russell, E. Seidel, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, J. Shalf, and I. Taylor. Enabling Applications on the Grid: A Gridlab Overview. *International Journal of High Performance Computing Applications*, 17(4):449–466, 2003.

[14] W. Anderson, P. Briggs, C. S. Hellberg, D. W. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg. Early Experience with Scientific Programs on the Cray MTA-2. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC2003)(published electronically)*, page 46, Washington, DC, USA, 2003. IEEE Computer Society.

[15] Apache Software Foundation. APACHE Web Server. http://www.apache.org.

[16] Apache Software Foundation. JetSpeed. http://portals.apache.org/jetspeed-1/.

[17] A. Armando and L. Compagna. Abstraction-driven SAT-based Analysis of Security Protocols. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, LNCS 2919, pages 257–271. Springer-Verlag, 2004.

[18] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, L. J. Dennis Gannon, K. Kennedy, C. Kesselman, D. Reed, L. Torczon, , and R. Wolski. The GrADS project: Software support for high-level grid application development. *International Journal of High Performance Computing Applications*, 15(4), Winter 2001. available from "`http://hipersoft.cs.rice.edu/grads/publications_reports.htm`".

[19] A. Biere. `http://www.inf.ethz.ch/personal/biere/projects/limmat/`.

[20] F. S. Bing Li, Chao Wang. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):143–155, April 2005.

[21] W. Blochinger, W. Westje, W. Küchlin, and S. Wedeniwski. ZetaSAT – Boolean satisfiability solving on desktop grids. In *Proc. of the IEEEInternational Symposium on Cluster Computing and the Grid (CCGrid 2005)*, volume 2, pages 1079–1086, Cardiff, UK, 2005.

[22] BlueHorizon. `http://www.npaci.edu/BlueHorizon/`.

[23] M. Boyd and T. Larrabee. A scalable, loadable custom programmable logic device for solving boolean satisfiability problems. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–21, 2000.

[24] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–955, 1998.

[25] J. Brevik, D. Nurmi, and R. Wolski. Predicting bounds on queuing delay for batch-scheduled parallel machines. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 110–118, New York, NY, USA, 2006. ACM Press.

[26] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of SuperComputing 2000 (SC'00)*, page 60, November 2000.

[27] W. Chrabakh and R. Wolski. GrADSAT: A Parallel SAT Solver for the Grid. Technical Report 2003-05, UCSB, March 2003.

[28] W. Chrabakh and R. Wolski. GridSAT: A chaff-based Distributed SAT solver for the Grid. In *Supercomputing Conference, Phoenix, AZ*, page 37. ACM, November 2003.

[29] Condor home page – `http://www.cs.wisc.edu/condor/`.

[30] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[31] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, page 181, Washington, DC, USA, 2001. IEEE Computer Society.

[32] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, London, UK, 1998. Springer-Verlag.

[33] A. Dandalis and V. K. Prasanna. Run-time performance optimization of an FPGA-based deduction engine for SAT solvers. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4):547–562, 2002.

[34] Data Star. `http://www.npaci.edu/DataStar/`.

[35] S. G. David A. Plaisted. A structure-preserving clause form translation. *Source Journal of Symbolic Computation archive*, 2(3):293 – 304, September 1986.

[36] M. Davis, G. Logeman, and D. Loveland. A machine program for theory proving. In *Communications of the ACM*, volume 5, pages 394–397, 1962.

[37] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, pages 201–215, 1960.

[38] M. den Burger, T. Kielmann, and H. E. Bal. TOPOMON: A monitoring tool for grid network topology. In *International Conference on Computational Science (2)*, pages 558–567, 2002.

[39] K. A. S. Fadi A. Aloul, Igor L. Markov. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Design Automation Conference*, pages 836–839. ACM/IEEE, June 2003.

[40] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM Press.

[41] S. L. Forman and A. M. Segre. NAGSAT: A Randomized, Complete, Parallel Solver for 3-SAT. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT2002)*, pages 236–243, 2002.

[42] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[43] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[44] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advanced reservation and co-allocation. In *International Workshop on Quality of Service*, 1999.

[45] G. F. Francine Berman and T. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., 2003.

[46] Gnutella. `http://www.gnutellanews.com`, 2001.

[47] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002.

[48] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, March 2002.

[49] P. Grace, G. Coulson, G. Blair, L. Mathy, W. Yeung, W. Cai, D. Duce, and C. Cooper. Gridkit: Pluggable overlay networks for grid computing. In *Proceedings Distributed Objects and Applications (DOA'04)*, pages 1463–1481, October 2004.

[50] S. Graham and B. Murray. http://docs.oasis-open.org/wsn/2004/06/wsn-ws-basenotification-1.2-draft-03.pdf, 2004.

[51] Grid Application Development Software (GrADS). `http://hipersoft.cs.rice.edu/grads`.

[52] H. E. Guy Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[53] E. A. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. In *PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg*, 2001.

[54] S. ichi Minato. *Binary decision diagrams and applications for VLSI CAD*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[55] K. Iwama. Cnf satisfiability test by counting and polynomial average time. *SIAM Journal on Computing*, 18(2):385–391, 1989.

[56] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 14–25, New York, NY, USA, 2000. ACM Press.

[57] P. James-Roxby, P. Schumacher, and C. Ross. A single program multiple data parallel processing platform for fpgas. *Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, 00:302–303, 2004.

[58] B. Jurkowiak, C. M. Li, and G. Utard. Parallelizing Satz Using Dynamic Workload Balancing. In *Proceedings of Workshop on Theory and Applications of Satisfiability Testing (SAT'2001)*, pages 205–211, June 2001.

[59] H. Kautz and B. Selman. Planning as satisfiability. In *ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[60] W. Kunz and D. Stoffel. *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Techniques*. Kluwer Academic Publishers, Boston, 1997.

[61] T. Larrabee. Efficient generation of test patterns using boolean difference. In *Proceedings International Test Conference*, pages 795–802. IEEE Computer Society, August 1989.

[62] T. Larrabee. Test pattern generation using boolean satisfiability. In *IEEE Transactions on Computer-Aided Design*, pages 4–15, January 1992.

[63] S. Larson, C. Snow, M. Shirts, and V. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2002.

[64] C. M. LI. A constrained-based approach to narrow search trees for satisfiability. In *Information processing letters 71*, pages 75–80, 1999.

[65] M. W. M. Lintao Zhang, Conor F. Madigan and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *International Conference on Computer Aided Design (ICCAD)*, pages 279–285, 2001.

[66] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, page 189, Washington, DC, USA, 1998. IEEE Computer Society.

[67] M. Moskewicz. `http://www.ee.princeton.edu/~chaff/index1.html`.

[68] B. D. Martino, J. Dongarra, A. Hoisie, L. T. Yang, and H. Zima. *Engineering The Grid: Status and Perspective.* American Scientific Publishers, 2006.

[69] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, 1994.

[70] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[71] G.-J. Nam, F. Aloul, K. Sakallah, and R. Rutenbar. A comparative study of two boolean formulations of fpga detailed routing constraints. In *ISPD '01: Proceedings of the 2001 international symposium on Physical design*, pages 222–227, New York, NY, USA, 2001. ACM Press.

[72] J. Novotny. The grid portal development kit. *Concurrency and Computation: Practice and Experience*, 14(13-15):1129–1144, 2002.

[73] OASIS. Web services reosurce framework (wsrf) tc. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf, 2003.

[74] "OMG". The complete formal/98-07-01: The corba/iiop 2.2 specification, 1998.

[75] J. Plank, M. Beck, and W. Elwasif. IBP: The internet backplane protocol. Technical Report UT-CS-99-426, University of Tennessee, 1999.

[76] F. Plasil and M. Stal. An architectural view of distributed objects and components in CORBA, Java RMI, and COM/DCOM. *Software-Concepts & Tools*, 19(3):14–28, 1998.

[77] C. Plessl and M. Platzner. Instance-Specific Accelerators for Minimum Covering. *J. Supercomput.*, 26(2):109–129, 2003.

[78] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[79] S. Radziszowski. Small Ramsey Numbers. *Electronic Journal of Combinatorics, Dynamic Survey DS1*, page 28, 1994.

[80] S. Reda and A. Salem. Combinational equivalence checking using boolean satisfiability and binary decision diagrams. In *Proceedings of the conference on Design, automation and test in Europe*, pages 122–126. IEEE Press, 2001.

[81] G. F. B. Riemann. ber die Anzahl der Primzahlen unter einer gegebenen Grsse. *Monatsber. Knigl. Preuss. Akad. Wiss. Berlin*, pages 671–680, November 1859.

[82] M. Ripeanu and I. T. Foster. Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 85–93, London, UK, 2002. Springer-Verlag.

[83] M. Ripeanu, A. Iamnitchi, and I. Foster. Performance predictions for a numerical relativity package in grid environments. *Int. J. High Perform. Comput. Appl.*, 15(4):375–387, 2001.

[84] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2002.

[85] S. Saroiu, P. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, pages 156–170, 2002.

[86] SAT 2002 benchmarks. `http://www.satlive.org/SATCompetition/2002/submittedbenchs.html`.

[87] SAT 2002 challenge benchmark. `http://www.ececs.uc.edu/sat2002/sat2002-challenges.tar.gz`.

[88] SAT 2002 Competition. `http://www.satlive.org/SATCompetition/`.

[89] SAT Competition. `http://satlive.org/SATCompetition/`.

[90] M. H. Schulz and E. Auth. Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification. *IEEE Transactions on ComputerAided Design*, 8(7):811816, July 1989.

[91] D. Schuurmans and F. Southey. Local search characteristics of incomplete sat procedures. *Artif. Intell.*, 132(2):121–150, 2001.

[92] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 337–343, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[93] SETI@home. `http://setiathome.ssl.berkeley.edu`, March 2001.

[94] M. Siekkinen, G. Urvoy-Keller, E. W. Biersack, and T. En-Najjary. Root cause analysis for long-lived tcp connections. In *CoNEXT'05: Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 200–210, New York, NY, USA, 2005. ACM Press.

[95] J. M. Silva and K. Sakallah. Grasp - a new search algorithm for satisfiability. ICCAD. IEEE Computer Society Press, 1996.

[96] J. P. M. Silva. Search Algorithms for Satisfiability Problems in Combinational Switching Circuits. Ph.D. Thesis, The University of Michigan, 1995.

[97] C. Sinz, W. Blochinger, and W. Kuchlin. PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. In *Proceedings of SAT2001*, pages 212–217, 2001.

[98] I. Skliarova. Reconfigurable hardware sat solvers: A survey of systems. *IEEE Trans. Comput.*, 53(11):1449–1461, 2004. Member-Antonio de Brito Ferrari.

[99] I. Skliarova and A. B. Ferrari. A software/reconfigurable hardware SAT solver. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(4):408–419, 2004.

[100] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 141–148, New York, NY, USA, 1998. ACM Press.

[101] L. L. Steve Tuecke and S. Meder. http://docs.oasis-open.org/wsrf/2005/03/wsrf-ws-basefaults-1.2-draft-04.pdf, 2005.

[102] T. Suzumura, S. Matsuoka, H. Nakada, and H. Casanova. Gridspeed: A web-based grid portal generation server. In *HPCASIA '04: Proceedings of the High Performance Computing and Grid in Asia Pacific Region, Seventh International Conference on (HPCAsia'04)*, pages 26–33, Washington, DC, USA, 2004. IEEE Computer Society.

[103] M. Swany and R. Wolski. Building performance topologies for computational grids. *Int. J. High Perform. Comput. Appl.*, 18(2):255–265, 2004.

[104] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[105] The TeraGrid Home Page. `http://www.teragrid.org`.

[106] TeraGrid. `http://www.teragrid.org/`.

[107] The BlueGene/L Team. An overview of the BlueGene/L Supercomputer. In *IEEE conference on supercomputing (SC2002)*, pages 1–22, 2002.

[108] The Great Internet Mersene Prime Search (GIMPS). `http://www.mersenne.org/`, 2001.

[109] S. S. Vadhiyar and J. J. Dongarra. Gradsolve: a grid-based rpc system for parallel computing with application-level scheduling. *J. Parallel Distrib. Comput.*, 64(6):774–783, 2004.

[110] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[111] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):645–662, /2001.

[112] Z. Website. `http://www.zetagrid.net/`, 2006.

[113] J. Widmer, C. Boutremans, and J.-Y. L. Boudec. End-to-end congestion control for tcp-friendly flows with variable packet size. *SIGCOMM Comput. Commun. Rev.*, 34(2):137–151, 2004.

[114] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running everyware on the computational grid. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 6, New York, NY, USA, 1999. ACM Press.

[115] R. Wolski, J. Brevik, G. Obertelli, N. Spring, and A. Su. Writing programs that run everyware on the computational grid. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1066–1080, 2001.

[116] R. Wolski, N. Spring, and J. Hayes. Predicting the cpu availability of time-shared unix systems on the computational grid. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[117] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999.

[118] T. W. W. S. A. working group. public draft. http://www.w3.org/TR/2003/WD-ws-arch-20030808/, August 2003.

[119] J. L. XL Jin. Multiagent SAT (MASSAT): Autonomous Pattern Search in Constrained Domains. In H. Y. et. al, editor, *Third International Conference on Intelligent Data Engineering and Automated Learning (IDEAL2002)*, pages 318–328, Manchester, UK, August 2002.

[120] The zChaff Satisfiability Solver. `http://ee.princeton.edu/~chaff/zchaff.php`.

[121] H. Zhang. SATO: An Efficient Propositional Prover. In *CADE-14: Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275, London, UK, 1997. Springer-Verlag.

[122] H. Zhang and M. Bonacina. Cumulating search in a distributed computing environment: A case study in parallel satisfiability. In *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO-94)*, pages 422–431, September 1994.

[123] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 295–313, London, UK, 2002. Springer-Verlag.

[124] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE Journal on Selected Areas in Communications*, volume 22, pages 41– 53, 2003.

[125] Y. Zhao, S. Malik, M. Moskewicz, and C. Madigan. Accelerating boolean satisfiability through application specific processing. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 244–249, New York, NY, USA, 2001. ACM Press.