Solving "Hard" Satisfiability Problems Using GridSAT

Wahid Chrabakh
and
Rich Wolski
Department of Computer Science
University of California Santa Barbara
{*chrabakh,rich*}*@cs.ucsb.edu*[1]

Abstract

We present an improved version of GridSAT, a distributed and complete satisfiability solver. The solver is used to solve a set of "hard" and previously unsolved set of satisfiability problems.

We enhance the GridSAT solver, which is based on the sequential solver Chaff, in two main ways. First the improved GridSAT solver uses immediate integration of shared clauses to achieve better solver performance. Second GridSAT's scheduler is enhanced to reduce communication overhead. Using a set of experiments, we show that GridSAT is capable of using simultaneously a wide variety of resources which range from shared desktop machines to supercomputers. GridSAT's scheduler was able to manage this diverse and dynamic resource pool for up to a month during some experiments to solve previously unsolved problems from the SAT 2002 [43] and the SAT 2003 [46] competitions. We present the improvements we made and the new results obtained using GridSAT.

Keywords: Parallel, Distributed, Scheduling, Satisfiability, Computational Grid.

1 Introduction

Grid computing [25] is an emerging field in computer science which main focus is the aggregation of geographically distributed and federated computational resources. The aggregated resources can be used by Grid applications to solve problems in science and engineering [33, 3, 40] which require large computing power. Solving such challenging problems and enabling new scientific results is an integral part of the grid computing vision.

One such challenging problem is propositional satisfiability. This problem involves finding a set of binary assignments which satisfies a set of constraints. The problem of solving satisfiability instances is important from both theoretical and practical perspectives. Satisfiability is theoretically significant because it was the first problem to be proven NP-complete [14]. Therefore, an algorithm for solving general instances of satisfiability is very likely to be compute intensive because it may take exponential time. An efficient method for solving satisfiability can also be used to solve other theoretical problems because these problems [24, 35] can be transformed to satisfiability instances. In practice, many engineering disciplines require the solution of domain specific instances of satisfiability. Such disciplines include scheduling [8], model checking [4], security [2], Artificial Intelligence [28] and software verification [26]. Satisfiability is especially important in the area of Electronic Design Automation (EDA). EDA encompasses a variety of problems such as circuit design [49], Field-Programmable Gate Arrays (FPGA) detailed routing [37], combinational equivalence checking [29, 39] and, automatic test and pattern generation [31].

There has been extensive research effort geared towards the development of gradually more efficient satisfiability solvers [36, 21, 23, 7]. These solvers use different techniques to navigate the entire search space. Modern solvers use optimizations which permit discarding parts of the search space during execution.

However, most modern solvers [36, 21, 23, 7] are sequential and are focused on improving existing algorithms and related heuristics. Fewer parallel solvers such as [12, 27, 50, 18] exist, and even fewer of the parallel solvers use an optimization termed *learning*. Learning (discussed in detail in Section 2.2) 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 learning solver daunting. As a result the best known solvers (in terms of speed and solution power) have until recently been sequential. Effective sharing of the proposition database by all processors working on a SAT instance is an important goal for a parallel SAT solver which uses learning.

These sequential solvers are characterized by heavy use of compute power (CPU) as well as the memory of the host

[1]This work was supported by grants from the National Science Foundation, numbered CAREER-0093166, EIA-9975020, ACI-0103759, and CCR-0331654.

machine. When the host's memory is exhausted the solver's progress becomes very slow. This happens when the problem is perceived as being "hard" by a solver. In the rest of this paper, we use the term "hard" or "complex" to refer to problems which use a long time to solve. We also use the term "easy" to describe those problems which are solved quickly by the solver under consideration. A SAT problem may be perceived differently by two SAT solvers. For example while a solver may find a problem hard and take a long time to solve a given problem. Another solver may find the same problem easy and is able to solve the problem quickly. However, there are SAT problems which are considered hard by all existing SAT solvers. Satisfiability instances, even those generated in practice, can be arbitrarily complex and difficult to solve by existing solvers. Actually, some of these problems including the ones we consider in this paper are left unsolved. Previous implementations of parallel solvers show that better performance is obtained when a large pool of computational resources is used. The aggregate CPU power and memory of the hosts make it possible to navigate the search space faster.

It is still an open research question to decide when using more resources increases the solver's performance. But research in parallel solvers, shows that using more resources can be (but not always is) a performance booster. Thus a Computational Grid populated by a a large pool of resources offers potential improvements in solver speed. These speed improvements can also enable a parallel SAT solver to solve previously unsolved problems which would have otherwise taken prohibitively longer durations to solve using a sequential solver.

Our previous work with GridSAT [12] shows that by dynamically acquiring resources as predicted by the GridSAT scheduler will benefit the solution to a particular SAT instance. In fact, GridSAT outperforms the best-known solver on all problems that this leading solver can complete. At the same time, GridSAT uniquely has been able to solve several previously unsolved problems using non-dedicated, wide-area Grid resources. Thus, by using Grid resources effectively, GridSAT constitutes a speed improvement over the fastest-known technique and has achieved new scientific results that have not previously been possible. In this work, we describe new results we have obtained with GridSAT. By combining different batch-controlled super-computers with interactive workstations and user desktop machines, we have applied GridSAT to hard SAT problems – ones that are not only unsolved but previous attempts to solve using other solvers have failed.

The resources in a computational grid may be of two different types: time-shared or batch controlled. In the case of time-shared resources the application will compete with other user applications running simultaneously on the host machine. However, since these resources are always available the application can continue to make progress. Other resources which are controlled by a batch scheduler, will participate intermittently in the application through some of their nodes. But these systems will provide significant compute power depending on the size of the application's request. This pattern of combining different types of resources is different from most existing implementations of parallel SAT solvers [27, 50]. In fact, there are many legacy applications in high performance computing which are limited to just using a particular resource at a time.

In order to enable a grid implementation of a SAT solver to use many resources simultaneously, we need to address two types of challenges. First the solver's algorithm needs to be modified so that it can run in parallel while ensuring that the parallel components cooperate to improve over-all efficiency. The second challenge is developing a framework capable of running the parallel solver in a very volatile computational environment.

We have developed GridSAT, a distributed satisfiability solver capable of running on a Computational Grid. GridSAT implements a parallel algorithm for solving satisfiability problems based on Chaff [36]. GridSAT allows resources to communicate in order to improve the solver's efficiency. The parallel implementation of GridSAT has been shown in previous experiments [12, 11] to achieve better performance than its sequential counterpart. GridSAT was also capable of solving problems which were previously unsolved.

In this paper, we will discuss further improvements we made to the GridSAT system. These improvements have made it possible to solve previously unsolved satisfiability problems from the field of FPGA [37] routing as well as artificially generated instances.

The paper is organized as follows. Section 2 introduces the basic SAT solver algorithm and some of the more advanced techniques used in modern solvers. In section 3 we present GridSAT's parallel version of the algorithm and the improvements added over previous implementations. We present experimental setup and results in section 4. Finally, we discuss related work in section 5 and conclude in section 6.

2 GridSAT Solver

A satisfiability problem is expressed as a boolean formula over a set of variables. Most solvers operate on formulas expressed in Conjunctive Normal Form (CNF). A CNF is a conjunction (logical AND) of *clauses*. A clause is an injunction (logical OR) of *literals*. A literal is either an instance of a variable (V) or its complement($\sim V$). A problem is called satisfiable if there exists a set of variable assignments that makes the formula evaluate to *true*. If such an assignment does not exist the the problem is declared *unsatisfiable*. The CNF has two important properties: any boolean formula can be algebraically converted to CNF, and for the original formula to be satisfiable all constituent clauses must be satisfiable.

GridSAT is based on Chaff [36], a sequential SAT solver. In Chaff, as well as other solvers, the performance of the al-

gorithm is enhanced by using techniques for adding new deduced clauses. In this section we explain the basic algorithm and how new clauses are generated.

2.1 The Basic Algorithm

The basis of Chaff and many modern SAT solvers is the Davis-Putnam-Logeman-Loveland (DPLL) [17] algorithm. Figure 1 shows a simplified flow chart describing the algorithm. This algorithm and its derivatives belong to the family of "complete" solvers that are guaranteed to find an instance of satisfiability if the problem is satisfiable, or to terminate once a sufficient set of all possible variable assignments have been examined proving that the problem is unsatisfiable. Variables can be assigned the values *true* or *false* but they are all marked as *unknown* initially. The algorithm uses heuristics to assign values to variables speculatively, but in an order that is likely to yield a truth assignment quickly if one exists. The speculative assignment of values to variables is called a *decision*. Because decisions are speculative (and may be undone) and because decisions have deductive implications, they are maintained as a stack. Each decision has a unique *level* in the *decision stack* with the first level in the decision stack containing variable assignments necessary for the problem instance to be satisfiable. For example, variables in clauses composed of a single literal will be added to this level. Other variables will be deduced to have a specific value and will be added to the first level as the algorithm progresses.

After each new decision, the algorithm searches for *unit clauses*. A unit clause is a clause with only one literal (i.e. a variable or its complement), without a determined truth value (*unknown*) and having the remaining literals all set to *false*. In a unit clause, the last remaining literal must have the value *true* for the clause to be *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. Even though an implication is a direct result of the previous assignment, it is also predicated on some of the previous variable assignments.

In DPLL a variable assignment (i.e. a variable is assigned a truth value) occurs when a decision is made or a variable is implied. *Boolean Constant Propagation* (BCP) is the procedure where the algorithm inspects the clause *database* in search of unit clauses, after each variable assignment. We use the term *database* in the rest of this paper 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, all clauses with a particular literal occurrence are indexed using a list of pointers. BCP is the most costly operation and accounts for up to 90% of the runtime [36].

When a decision is made, resulting implications are added to the current decision level. More implications might be added in a cascade because of earlier implications. This process continues until no more implications are found or contradicting assignments to the same variable are detected. In the case when there are no more implications and if not all clauses are satisfied, a new decision is made as shown in figure 1. When a new decision is made an additional decision level is added to the decision stack. In the other case where a contradiction happens,the algorithm has encountered a *conflict*. A conflict occurs when the implied variable was previously assigned the opposite value. When a conflict happens the algorithm resolves the contradiction when possible before proceeding. In order to remedy a conflict, a simple approach is to flip the value of the previous decision and then try again. In case when a decision has been tried both ways, the first previous decision which can be flipped is tried. If the algorithm cannot find a previous decision which was not tried both ways then the problem is found to be unsatisfiable. This method is slow and may require trying all 2^N combinations of variable assignments when the problem is unsatisfiable, where N is the number of variables. More sophisticated conflict analysis techniques are presented in the next section. The conflict analysis procedure points to a level in the decision stack to which the algorithm can back-jump. Non-chronological back-jumping [65] occurs if the algorithm jumps by more than one decision level. After back-tracking the algorithm continues by making a new decisions or deducing new implications.

Eventually the algorithm terminates under one of two possible conditions. If the problem is satisfiable, a set of variable assignments which result in all clauses evaluating to *true* is found. This termination condition occurs when all clauses are satisfied because of the current set of variable to assignments. Note that not all variables need to be assigned a truth value for this to happen. The problem is deemed unsatisfiable if the algorithm backtracks completely to the first decision level and there is a conflict due to deduced variable assignments at this level. Since the variable assignments at this level are necessary for the problem to be satisfiable then this is a conflict that the algorithm cannot resolve. Therefore, the algorithm concludes that the formula is unsatisfiable.

2.2 Conflict Analysis and Learning

A more sophisticated and effective method to do conflict analysis is *Learning*. Learning [47, 30, 48] is the augmentation of the initial formula with additional implicate clauses. These new clauses indicate search spaces which were found to have no solution because they result in conflicts. The presence of these clauses restricts the search space and prevents the solver from retrying those parts of the search tree. Learned clauses represent redundant information because they can be deduced from the initial set of clauses. Thus learned clauses can be discarded without effecting the satisfiability of the solution.

In DPLL with learning new implicate clauses are deduced due to a conflict. Conflict analysis is based on implication graphs. An *implication graph* is a DAG which expresses the implica-

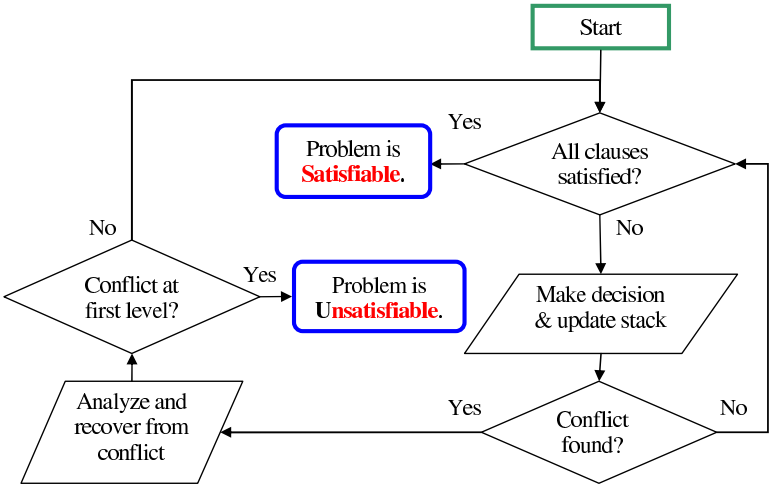

Figure 1: Flow chart for the DPLL algorithm

tion relationships of variable assignments. 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 clause that caused its implication. That is, the clause that has previously become a unit clause and caused this variable to be implied (i.e. 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 indexes greater than 1, thus we use clause 0 (which does not exist) as antecedent for decision variables.

A *learned clause* is obtained by partitioning the implication graph into two sides. One partition called the *reason* and contains all the decision variables. The other partition which contains the conflict is called the *conflict* side. Different learning schemes are generated from different partitioning methods. 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.

The purpose of the new clause is to prevent, in the future, the set of simultaneous assignments which led to the current conflict. The new learned clause is obtained by using the complement of the variables on the reason side with edges intersecting the cut. In addition the conflict clauses cause the solver to perform a non-chronological backtrack. After backtracking, the new decision level is the highest decision level among all the decision levels of the variables in the new learned clause. Chaff [36] uses a method called FirstUIP. This method is based on finding a *dominant node* to the conflict nodes defined as a node where all paths from the current decision to the conflict pass through. The variable corresponding to the selected dominant node is the only variable added to the learned clause which is not a decision variable. 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 site are on the conflict side. For a more detailed explanation of the algorithm please refer to [12, 11, 36, 48].

During execution, the algorithm learns a huger number of clauses. These clauses consume a lot of memory and may overwhelm the capacity of the host. All learned clauses represent redundant information and may be deleted. However, if a learned clause is an antecedent it cannot be removed since it is essential to the coherence of the algorithm. Clauses which are not antecedents can be removed without affecting the correctness of the solver. Chaff implements heuristics to select which learned clauses are deleted depending on their size and other properties. Deleting some of the learned clauses periodically alleviates memory use and allows the addition of new learned clauses which are currently more relevant.

3 GridSAT: SAT Solver for the Grid

The initial GridSAT implementation was used to prove the feasibility and efficiency of a parallel and distributed solver. It was shown in [12, 11] that GridSAT is faster and even solves some previously unsolved problems. Since satisfiability problems can be arbitrarily complex, there are a set of problems which we were not able to solve. In this section we present GridSAT's main features and point-out the improvements that were introduced to increase its efficiency.

GridSAT's distributed solver addresses three significant challenges. First, GridSAT parallelizes the search algorithm that is navigating the space of possible truth assignments. Second, certain learned clauses from the various solvers are distributed and shared across Grid resources. Finally, the GridSAT application components are dynamically scheduled so that they may take advantage of the best possible resources at the time

and they can be used profitably by the algorithm.

SAT problems vary in terms of their resource requirements. The two main resources which affect solver performance are CPU speed and memory size. CPU speed evidently makes execution faster. Available memory, however, is used to store learned clauses. The role of these clauses is to restrict the search space. But since even problems from practical application have enormous search spaces, a fast CPU with little memory will result in extremely slow progress. The memory size is therefore crucial for good solver performance. In case where the memory is exhausted by storing antecedents, the solver's progress is inhibited significantly. Using a given memory size and CPU speed, a satisfiability problem could be perceived by a particular solver as being *easy* or *hard*. Some problems are considered *easy* because they can be solved using one CPU in a short time span. Other problems however are *hard* because they require many CPUs and a long time period. A parallel SAT solver which dynamically adjusts to the problem's requirements, can achieve better performance and more efficiently use the available resource.

To apply a parallel search technique to SAT, we split the problem at hand into subproblems (having decision stacks with different truth assignments), each of which is independently investigated for satisfiability. Subproblems, themselves, may be split in the same way, forming a recursive tree, each node of which is assigned to a logically distinct processor. A subproblem represents part of the search space. Clause sharing is facilitated by identifying the important clauses relevant to each side of a split, and by eliminating clauses from the clause database pertaining to each side.

The goal of GridSAT is to keep the execution as sequential as possible and to use parallelism only when it is needed. Because problem difficulty is unpredictable and parallelism overhead could be high, GridSAT attempts to add resources (machines with sizable memory) only when the current resource set (which starts with one machine) becomes overloaded.

3.1 Parallelizing SAT

GridSAT acquires new resources when existing sub-problems are split into two sub-problems covering disjoint, but complementary, parts of the original search space. For GridSAT the split process modifies the current problem and spawns a new one as shown in Figure 2. The left part of figure 2 shows the old decision stack of process A before splitting. This process (also called *client* in GridSAT parlance) was assigned a subproblem and is now splitting its search space with client B. The right part of figure 2 shows the modified problem stack for client A and the newly created problem stack for client B after splitting. The first decision variable in the second decision level of Client A's original stack is the pivotal point in the split. Clients A and B assume two different values for this variable. Since this variable is given a specific value in both clients, then it becomes part of the first decision level in both cases. For client A, all implications which were previously in the second decision level are now also part of the first decision level of the modified decision stack. Therefore, Client A's new decision stack is created by making all variables on the second decision level of the assignment stack part of the first decision level. The newly generated problem stack for client B consists of a set of variable assignments and a set of clauses. The variable assignments include all assignments from the first decision level and the complement of the first assignment in the second decision level of Client A's original stack. Thus insuring the splitting of the search space.

After splitting, each process maintains its own separate clause database. In order to alleviate memory usage, inconsequential clauses are removed. A clause is removed from a client's database when it evaluates to *true* because of the assignments made at the first level of its decision stack as a result of the split. Actually, inconsequential clauses are removed after every time the first decision level is augmented.

A notable 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 may be slower than if it were executed sequentially. This problem is occasionally referred to as the "ping-pong" effect [27].

In the following sections we will detail newly introduced modifications to the splitting procedure and clause sharing to help improve the overall solver performance. These improvements include several aspects:

- A new method for using shared clauses
- Adaptive clause sharing
- Reduction of communication overhead during problem transfer

3.2 Sharing and Distributing the Clause Database

Each GridSAT process is assigned a part of the search space disjoint from the search space of all other processes. This is insured by giving each process a unique top decision level in the stack. This level may be augmented but is never reduced. Because of the uniqueness of the stack, solvers will tend to make different decisions which in turn results in varying implications. Therefore, the learned clauses, which are dependent on the decision stack, as well as previous learned clauses, will most probably differ for various processes. Thus when these learned clauses produced by one client are shared with other clients they help prune parts of their search space

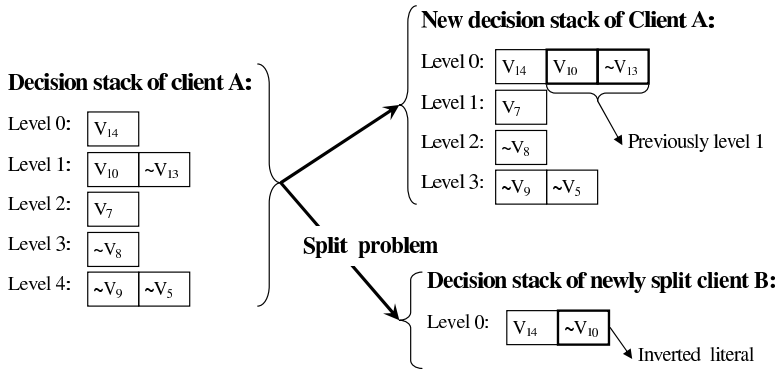

Figure 2: Example of stack transformation when a problem is split into two clients

which they have not yet investigated. 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, 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. Next we present and analyze the old and new methods GridSAT uses for sharing learned clauses. We explain the characteristics of each method and give the motivation for using the new method.

3.2.1 The Old Method for Sharing Clauses: In previous versions of GridSAT we limited the newly obtained clauses to only being merged into the clause database 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 in this case 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 will not be altered by subsequent decision.

Under the conditions outlined previously, merging a received clause is straight forward. 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, then variable corresponding to the *true* literal could only come from the first decision level. Since this variable will always be *true*, then 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, an 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 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. If a clause in the batch causes a conflict then the solver terminates immediately.

3.2.2 The New Method for Sharing Clauses: We made a very important observation while running different experiments especially with problems that were hard and took a very long time without making progress. We realized that when the problem was hard, all processes were not able to use the clauses received from other processes because none of them were able to achieve enough progress which will permit them to backtrack to the first level of the decision stack. Therefore all the shared clauses were being saved by local solvers wasting valuable memory space but were never used. Thus sharing clauses did not have the desired effect of helping to prune the search space of the local solver. Instead performance was degraded because of wasted memory space. The solution we implemented allows immediate integration of received clauses into the solver's clause database. The implementation of this solution is more complex compared with the algorithm above because the decision stack may contain multiple levels of speculative decisions.

The algorithm for merging clauses starts by inspecting the newly obtained clause. 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 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.

For performance considerations which will be mentioned later, the new method merges clauses in batches as well. When a new clause is merged, the decision stack is modified and a backtrack is performed in three of the five cases presented above. In addition, every backtrack reduces the stack depth unless of course 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 is also cleared so that the solver can reconstruct the resulting implications while taking the new clauses into consideration. When the solver backtrack to the first level in the decision stack, the new merging method becomes the same as the simpler previous method.

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. 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. By merging new clauses, more relevant search spaces are chosen based on new implications and conflicts and not by random chance.

As described in [50], the exact effect of sharing clauses is not yet known. In addition, 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 the solver at sometimes high rates, especially

if the number of processes used is high. Therefore merging the clauses immediately will cause 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, the solver is parametrized to allow clause merging only after a fixed number of iterations. This results in the clauses being merged in batches in a similar fashion to the old algorithm.

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. Checking each new clause received by a solver to insure 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 are 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 is a slim chance that duplicate clauses will become an overwhelming problem. In future work, we will instrument GridSAT to find out how much duplication really occurs for a given set of problems.

3.2.3 Dynamically Adjusting Size of Shared Clauses: GridSAT clients only share "short" clauses in order to minimize communication cost. Short clauses are expected to have a higher impact on pruning the search space and are more probable to generate implications. In fact the pruning effect of a clause is inversely proportional to its size (i.e. number of its literals). Previous GridSAT implementations take the maximum length of shared clauses as a static parameter.

Using a static value for determining the maximal size of shared clauses, may lead to one of two possible bad scenarios. First, if the value is too small the processes will not generate clauses smaller than the suggested value and no clause sharing will happen. In the second scenario, the used maximal clause size is low and causes a huge number of clauses to be shared. As a result, an influx of learned clauses may overwhelm the solvers with unnecessary communication and computational overhead. In addition, it is hard to determine a priory what the maximal clause size should be for a given SAT instance. In order to avoid both of these scenarios, the maximal clause size can be varied during the application execution using a given problem instance.

In the current implementation of GridSAT, the maximal size of shared clauses is determined dynamically. We set the absolute minimum for the maximal size to two. The maximal size of learned clauses is adjusted depending on a user supplied maximum rate of communication overhead due to clause sharing. The user can supply a maximal rate for shared clauses or use the default (set to 3). A process monitors the rate of shared clauses and calculates it periodically every five minutes. When this process notices that the maximal rate was exceeded, it broadcasts immediately an incremental decrease of the maximal clause size. This step insures that communication overhead resulting from shared clauses will only exceed its maximum for a short period of time. If the rate is below the maximal rate, then the monitoring process waits for half an hour before increasing the maximal rate and broadcasting the new value to the rest of the solvers. This allows the communication overhead to remain under its maximum value for a long time period. The user can also set an absolute maximal size for shared clauses.

3.3 GridSAT Architecture and Resource Scheduling

GridSAT is implemented as a special form of the master/client model where individual clients communicate directly and share clauses. The master consists of four main components. These are the resource manager, the client manager, the scheduler and the checkpoint server. A general architecture of the master process is shown is figure 3. External components with which the master interacts are shown in clouds.

The resource manager loads resource information from one or more Grid information systems such as Globus MDS [15] and the NWS [62, 51, 63]. The scheduler as shown in figure 3 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. For shared resources only one GridSAT process per host is launched. For batch systems, the resource manager launches one job at the start of the execution. Additional, jobs could be manually submitted and GridSAT will use their resources when they become available. Actually the client manager will accept 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 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 which 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. Some problems are solved easily using a single host after a short time period. Other problems, however, might be harder and require a large number of hosts and a longer time period. By starting with a small resource pool and expanding the set of used resources, GridSAT achieves two goals. First, a small number of resources will be used to solve the easy problems which results in a smaller communication overhead and therefore shorter time to solve the problem. Second, GridSAT can adapt resource usage to how difficult the problem is perceived. If at a particular stage the problem is perceived difficult the size of the resource pool used will grow. At another stage, the same problem might be perceived to be easy and a smaller resource set will be used, and excess resources

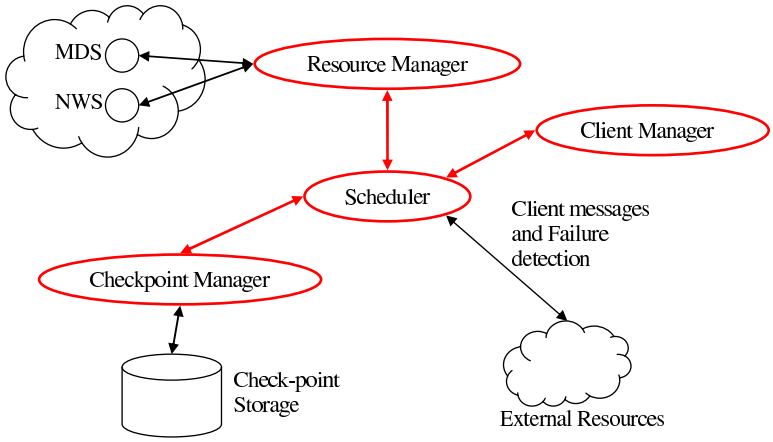

Figure 3: GridSAT components and their internal and external interactions. The external components and systems which GridSAT uses, such as the Globus MDS and the NWS, are shown in clouds.

will be released.

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 MDS [15] and NWS [62] or simply a configuration file. 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 [62, 51]. 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 notifies the master that it wants to split its assigned subproblem with another client when its memory usage exceeds a certain limit or after running for a specific 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.

The splitting process is performed by the cooperation of three components: the master, the splitting client and an *idle* client. The *idle* client is a process which was not previously assigned a sub-problem to investigate. Figure 4 shows the steps taken during the splitting process. Client A which has presumably been running using a sub-problem, has detected that it needs to split its problem. 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). In previous GridSAT implementations, message (3) is very large and varies in size from 10KB to 500MBytes. By using direct peer-to-peer communication the overall communication overhead is reduced. When the splitting is successfully completed, both clients 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. Check-pointing is discussed further in the next section. The scheduler can detect and recover from client failures during this procedure.

Message (3) above allows the transfer of a newly created subproblem 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 clause: It is the largest component and is 100s of MBytes in size.

3.3.1 Reducing Communication Overhead: In this version of GridSAT we have added two modifications to reduce the communication overhead of the solver. First, problem files are copied only once where several hosts share a common file system. Therefore split messages to the same set of hosts will be smaller since it will not include the second part mentioned earlier. 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 algorithm mentioned above. Using this methods the new client will not have to idly wait for the entire message to arrive before starting solving the newly assigned sub-problem. The old client still waits 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.

Finally the GridSAT solver terminates when all sub-problems have been solved or one the clients finds a satisfying assignment. In the latter case the client which 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 [26] where knowing all satisfiable instances is helpful. GridSAT can also enumerate all the instances where a problem is satisfiable. In all cases, when the master determines that the problem is solved, it sends a message to all clients requesting them to terminate.

3.4 Check-pointing System

The current version of GridSAT uses check-pointing to recover from failures. 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 for each client. In this case checkpoints for a client will be updated only when more variables are added to the first decision level.
- Heavy checkpoints: In addition to the light checkpoint data, we 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. This type of checkpoints can be saved at regular time intervals in addition to the instances when the top level is augmented.

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 [38]. 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 client are assigned problems through splitting only when all checkpoints are assigned to active clients. A user could potentially stop executing a problem at some moment in time to start another problem for example. The user can then resume solving the same problem using the saved set of checkpoints.

3.5 Problem Migration

GridSAT processes communicate as peers during problem splitting. Even after the implementation of the optimization presented above which reduce communication overhead, peer-to-peer messages are still the largest. Therefore, more efficient problem splitting help improve the overall solver's efficiency. More efficient problem splitting could be accomplished when clients belong to a pool of well connected resources. Such pools of resources are usually presented when new batch jobs reach the head of their waiting queue and start running. GridSAT migrates problems from dispersed nodes to processes which are part of a batch job.

The scheduler identifies batch processes in a static fashion using their host names. Instead of creating a new sub-problems through splitting with a remote node, the scheduler requests the remote node to migrate to the batch process. Using migration allows future splitting to happen between peers belonging to the same batch jobs. This leads to reduction in overall communication overhead. In future versions of GridSAT, determining when migration happens will be achieved through a more dynamic approach.

3.6 Efficient Use of Batch Jobs

Batch schedulers are usually used to control Supercomputing facilities [9, 53] and collections of grid resources such as Condor [13, 52]. 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 the total time the nodes in the batch job are assigned to his job. 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, which mostly use MPI [19], the number of processes spawned is sufficient to insure 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 possi-

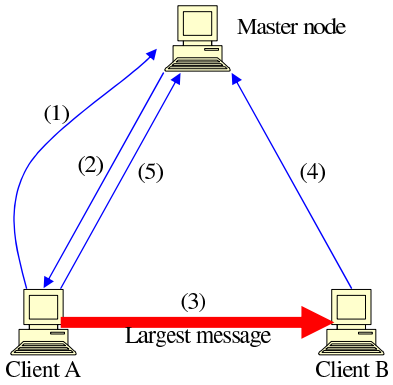

Figure 4: Communication scenario of splitting the subproblem assigned to client A with client B

ble to immediately split a sufficient number of sub-problems. This will lead to more efficient use of batch jobs but may 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 duration. 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. Remote GridSAT nodes, which are numerous, will migrate immediately to occupy batch nodes. After, migration takes place and since networks are fast within supercomputing nodes, splitting happens at higher rates especially after the above mentioned reductions in communication overhead. Actually 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.

4 Experimental Apparatus and Results

Since GridSAT is a true grid application 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 experiment we simultaneously use computational resources which belong to collections of individual machines, small size research clusters and supercomputing scale clusters. The computational resources we used are composed from four main sources:

- VGrADS [59] testbed with additional machines from the University of California, Santa Barbara (UCSB)
- Blue Horizon [9] located at the San Diego Supercomputer Center (SDSC)
- TeraGrid [58] site at the San Diego Super Computing Center (SDSC)
- TeraGrid [57] site at the National Center for Supercomputing Applications (NCSA)
- Data Star [16] supercomputer at SDSC

The TeraGrid [53] project is a national scale project which is aimed at building the worlds largest distributed infrastructure for open scientific research. It includes five sites at SDSC [58],NCSA [57], Argonne National Laboratory [54], Pittsburgh Super Computing center [56] and Caltech CACR [55]. Additional sites and resources are planned at Oak Ridge National Lab (ORNL); Purdue University, Indiana; Indiana University, Bloomington; and the Texas Advanced Computing Center (TACC) at The University of Texas at Austin.

The **V**irtual **Gr**id **A**pplication **D**evelopment **S**oftware (GrADS) project [59], a continuation of GrADS [5, 22] is a comprehensive research effort studying Grid programming tools and application development. GrADS includes a set of programming tools for managing grid applications using performance models. Scheduling applications in GrADS uses compiler pre-processing of the programs and introduced instrumentation combined with NWS data. The tools GrADS uses are included in a software package termed *GrADSoft*. To facilitate experimental application research and testing, the project maintains a nationally distributed grid of resources for use as a production testbed. The baseline Grid infrastructure is provided by Globus and the NWS, upon which is layered a set of programming abstractions. In this work we extend GridDSAT to use all resources that do not currently benefit from these sophisticated Grid programming tools. GridSAT components (i.e. master and client) use the EveryWare [60, 61] messaging system for communication.

During our experiments, none of the resources we used were dedicated to our use. The VGrADS testbed, the UCSB machines, and the supercomputing resources were all in continuous use by various researchers and application scientists at

the time of the experiment. 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. Sometimes we were requested to temporarily vacate some specific resources because some users wanted to run experiments without interference from other applications. We consider this to be a realistic scenario for Computational Grid computing, but it makes repeatable timings of similar problems (particularly those we ran for long periods) difficult. In particular, in batch controlled system such as Blue Horizon, Data Star and the TeraGrid, a user presents a request for a number of nodes and a maximum duration. After waiting in the job queue, the user's job runs with exclusive access to the nodes during execution, but the queue wait time incurred before execution begins is highly variable. However, the effect of resource contention is almost assuredly a performance-retarding one. 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.

In previous experiments [12] we showed how GridSAT can simultaneously use small clusters and a collection of lab machines in conjunction with high end supercomputers such as Blue Horizon. The experiments used a single job request on the Blue Horizon with a maximum timeout of 12 hours.

The set of experiments we present in this paper 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 SAT2002 conference [41] and SAT2003 benchmarks [44]. These benchmarks are used to judge and compare the performance of automatic SAT solvers at the SAT2002 [43] and SAT2003 [46] conferences. All the problems in the benchmarks are shuffled to insure 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* [42, 45]. 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. We only chose problems which are hard so that we can demonstrate the ability of the GridSAT system to solve such challenging problems. These problems were deemed hard by all participating solvers.

We investigate seven previously unsolved problems divided as follows:

- 3 instances from the SAT 2003 benchmark category,
- 4 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 this set of experiments, the resource pool included 40 machines from the VGrADS testbed and an additional machine (that we could completely instrument) as a master node. The machines were distributed among three sites: three clusters (separated by campus networking) at the University of TN, Knoxville (UTK), five desktop machines at the University of San Diego (UCSD) and ten machines from the MAYHEM [34] lab at the University of California, Santa Barbara. An additional node, designated the master node, was at UCSB. The machines had varying hardware and software configurations.

In these experiments 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 relaunched 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.

4.1 Results

The experimental results are summarized in Table 1. 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 and was later solved by a

solver, 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. In such cases, experiments could be continued using the saved checkpoints.

Table 1 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 still waiting in the queue. Thus when the problem got solved before they got to the head of the queue the batch jobs were canceled. On the other hand the second problem *k2fix-gr-rcs-w9.cnf* took much longer to solve, it took more than two weeks. We expect that some Grid applications will require running for such extended periods of time. Table 2 gives a more detailed description of the resource used during this experiment. For each job a number of GridSAT solver components were launched as indicated in the last column of table 2. The number of processes per node is determined so that each process gets a minimum of 1/2 GByte or 1GByte of memory. In table 3 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.cnf* was also solved using the VGrADS testbed only under similar circumstances to *k2fix-gr-rcs-w8.cnf*. We previously tried solving this problem in [12] 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.cnf*. 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. The figure shows several jobs from Blue Horizon, Data Star and TeraGrid sites participating in the execution. Note that the processor count is represented in logarithmic scale. 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. In figure 6 we show the total number of processes used by GridSAT during the same period. GridSAT was able to manage up to 350 processes running on different resources as show in this figure.

The satisfiability solver performs mostly integer, branching and load/store operations. The number of floating point operations is very low (less than .1 FLOPS). Floating point operations are only used to handle time related events. We present in figure 7 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 7 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 of case 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, 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 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. GridSAT was able to continue running experiments on the set of available resources through this transition. The scheduler would try to submit jobs but it would notice that the Blue Horizon resource was not responding. The failure of this single (but important) resource which did not affect the already running experiments shows

File name	Description	SAT/UNSAT/*	Time	GridSAT Result
3bitadd-31.cnf	theoretical	UNSAT	8 days	-
k2fix-gr-rcs-w8.cnf	FPGA Routing	*	83261 sec (23 hours)	UNSAT
k2fix-gr-rcs-w9.cnf	FPGA Routing	*	14 days and 8 hours	UNSAT
cnt10.cnf	Theoretical	SAT	13134 sec (4hours)	SAT
comb1.cnf	Model Checking	*	11 days	-
f2clk50.cnf	Model Checking	*	9 days	-
hanoi6.cnf	Theoretical	SAT	23 days	-

(*): problem solution initially unknown

Table 1: GridSAT results using VGrADS testbed, Blue Horizon, Data Star and TeraGrid. All these problems were not previously solved by any other solver.

Computational resource	Job count	Job duration(hours)	Number of nodes	processes/node
Blue Horizon	2	10	100	3
Blue Horizon	1	12	100	3
DataStar	2	10	8	11
TeraGrid @ SDSC	1	10	40	2
TeraGrid @ SDSC	1	12	40	2
TeraGrid @ SDSC	3	10	4	2
TeraGrid @ SDSC	4	5	4	2
TeraGrid @ NCSA	3	10	4	2
TeraGrid @ NCSA	4	5	4	2

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

Table 2: Batch jobs used to solve the k2fix-gr-rcs-w9.cnf instance from SAT 2003 benchmark

Computational resource	node-hours	CPUs/node	CPU-hours
Blue Horizon	3200	8	25600
Data Star	160	11	1760
TeraGrid @ SDSC	1080	2	2160
TeraGrid @ NCSA	200	2	400
GrADS testbed(*)	13750	1	13750

(*) machines were shared with other users

Table 3: CPU-hours per resource used to solve the k2fix-gr-rcs-w9.cnf instance from SAT 2003 benchmark

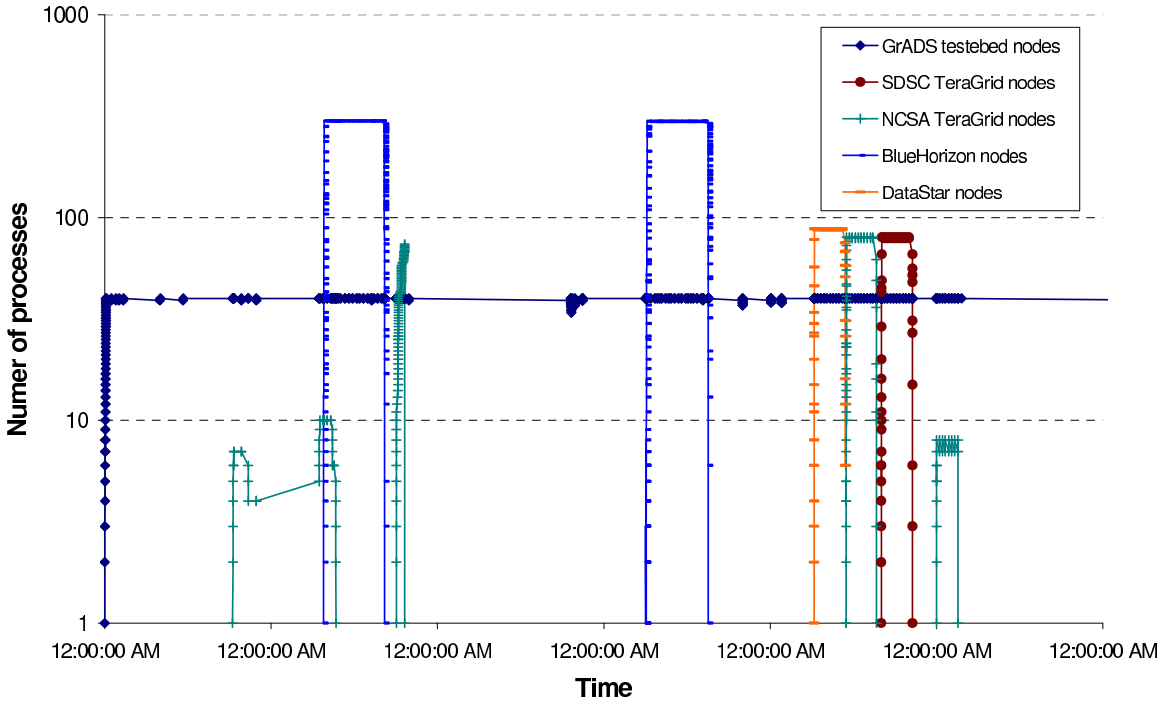

Figure 5: A six day snapshot representing GridSAT processor count usage from the different resources in logarithmic scale.

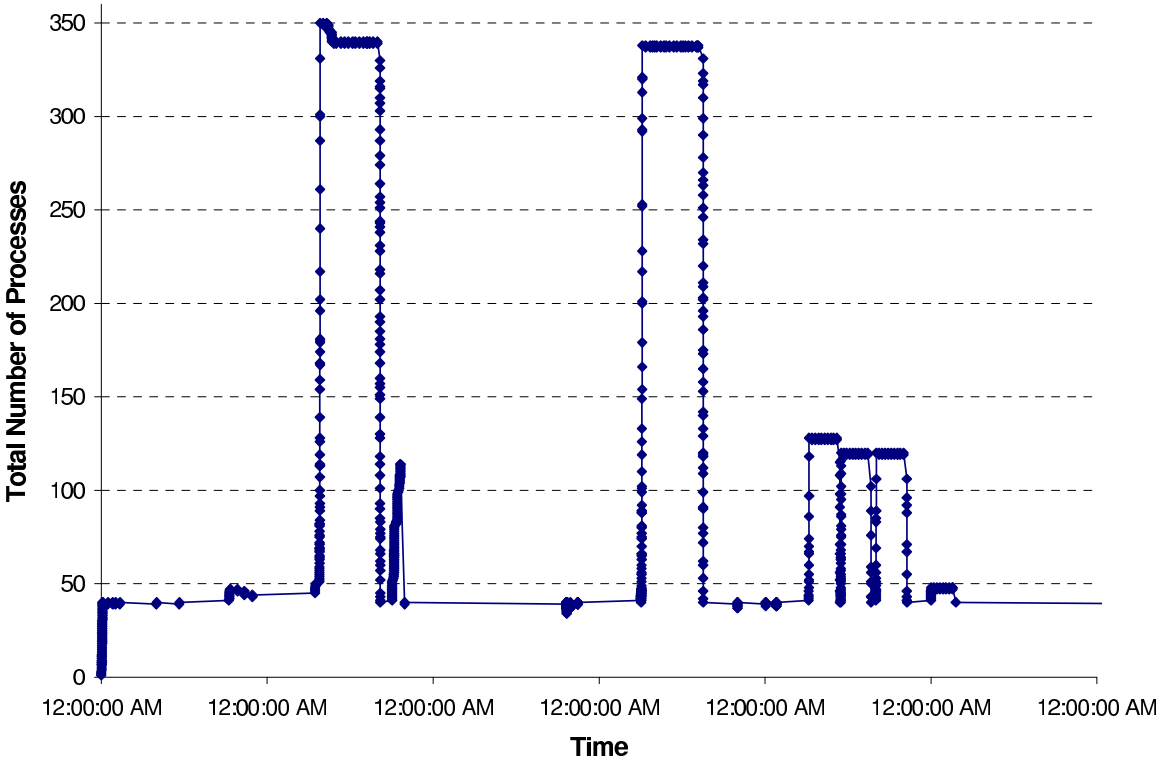

Figure 6: The total processor count usage from all the resources for the same six day snapshot shown in figure 5.

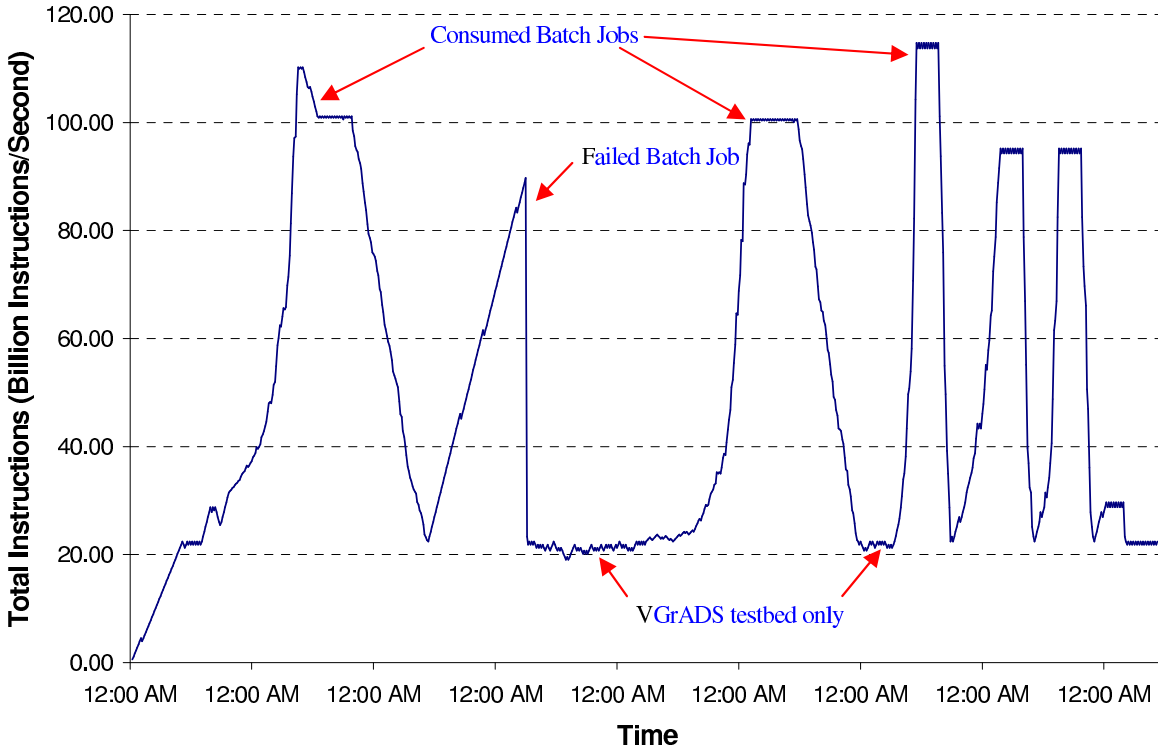

Figure 7: Estimation of Integer Operations per second usage for all resources during the same six day snapshot shown in figure 5.

the robustness of GridSAT.

5 Related Work:

This paper covers both parallel SAT solvers and master-client applications in a grid environment. We discuss related work in both of these areas.

There are several parallel solvers in the literature. PSATO [64] is based on the sequential solver PSATO. PSATO is concentrated on solving 3-SAT and open quasi-group problems. An other solver is Parallel SATZ [27] which is the parallel implementation of SATZ [32]. 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 [50] 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. GridSAT is designed to attack *easy* as well as *hard* problems which take extended periods of time.

A different approach is presented by NAGSAT [18]. 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. The solver is only applied to a set of randomly generated 3-SAT instances.

A parallel scheme based on a multiprocessor implementation is presented in [66]. The configurable processor core was augmented with new instructions to enhance performance. Data parallelism is used to speed-up execution of common functions in the DPLL algorithm. Unlike GridSAT, this approach relies on specific hardware.

In the area of Grid Computing there has been a great deal of research into the scheduling of master-worker applications [6, 10, 1, 20]. NetSolve [10] is dedicated to providing support for access to scientific libraries remotely. Nimrod-G [1] is targeted to the exploration of range of parameters for scientific applications. These master-client systems use a predefined number of clients with an established set of resources. This is not the case for GridSAT where the number of clients changes depending on the problem and uses any clients available. In addition, GridSAT is not a simple master-worker application. GridSAT clients cooperate and communicate as peers in order to exchange problems and newly derived clauses. Most of the above systems use a centralized scheduler. GridSAT uses a combination of a central scheduler and local client decisions to assign and split existing jobs. The satisfiability problem is different from most existing applications because it does not have a predictable runtime or resource needs. GridSAT runs for extended periods of time using a variety of resources.

6 Conclusion

This paper presents a new version of GridSAT which implements a parallel, distributed and complete satisfiability solver. In order to solve harder problems, new improvements to both the algorithm and architecture of GridSAT were introduced. GridSAT is capable of merging newly received shared clauses immediately to the clause database to improve the solver's efficiency. Also communication overhead is reduced by selectively sending important information first and avoiding redundancy when possible.

The experiments we presented show GridSAT's ability to manage and use a diverse set of dynamic computational Grid resources. The experiments lasted for weeks as a testament to the robustness of the application. During these experiments new previously unsolved problems from practical and theoretical fields were solved.

References

[1] D. Abramson, J. Giddy, I. Foster, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for Global Grid? In *The 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, 2000.

[2] L. C. Alessandro Armando. Abstraction-driven sat-based analysis of security protocols. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, pages 257–271, May 2003.

[3] 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.

[4] C. W. B. Li and F. Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. December 2003.

[5] 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`".

[6] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing 96, Pittsburgh, PA*, November 1996.

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

[8] R. Bjar and F. Many. Solving the Round Robin Problem Using Propositional Logic. AAAI/IAAI, 2000.

[9] BlueHorizon. http://www.npaci.edu/bluehorizon/.

[10] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 1997.

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

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

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

[14] S. A. Cook. The complexity of theorem-proving procedures. *Third Annual ACM Symposium on Theory of Computing*, 1971.

[15] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing*, 2001.

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

[17] M. Davis, G. Logeman, and D. Loveland. A machine program for theory proving. Communications of the ACM, 1962.

[18] S. L. Forman and A. M. Segre. Nagsat: A randomized, complete, parallel solver for 3-sat. SAT2002, 2002.

[19] M. P. I. Forum. Mpi: A message-passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, 1994.

[20] G. Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, University of California, San Diego, May 2001.

[21] 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.

[22] GrADS. `http://hipersoft.cs.rice.edu/grads`.

[23] 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.

[24] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addision Wesley, 1979.

[25] C. K. I. Foster and S. Tuecke. The anatomy of the grid. 2001.

[26] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. *International Symposium on Software Testing and Analysis*, 2000.

[27] 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.

[28] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 359–379, August 1992.

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

[30] T. Larrabee. Efficient generation of test patterns using boolean difference. pages 795–802.

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

[32] C. M. LI. A constrained-based approach to narrow search trees for satisfiability. Information processing letters 71, 1999.

[33] W. W. Li, R. W. Byrnes, J. Hayes, A. Birnbaum, V. M. Reyes, A. Shahab, C. Mosley, D. Pekurovsky, G. B. Quinn, I. N. Shindyalov, H. Casanova, L. Ang, F. Berman, P. W. Arzberger, M. A. Miller, and P. E. Bourne. The encyclopedia of life project: grid software and deployment. *New Gen. Comput.*, 22(2):127–136, 2004.

[34] MAYHEM home page – `http://pompone.cs.ucsb.edu//`.

[35] M. T. Michael R. Dransfield, V. Wiktor Marek. Satisfiability and computing van der waerden numbers. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, pages 1–13, May 2003.

[36] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: Engineering an Efficient SAT Solver. 38th Design Automation Conference (DAC2001), Las Vegas, June 2001.

[37] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar. A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints. *International Symposium on Physical Design (ISPD), Sonoma Wine County, California*, pages 222–227, 2001.

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

[39] 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.

[40] M. Russell, G. Allen, G. Daues, I. Foster, E. Seidel, J. Novotny, J. Shalf, and G. von Laszewski. The astrophysics simulation collaboratory: A science portal enabling community software development. *Cluster Computing*, 5(3):297–304, 2002.

[41] SAT 2002 benchmarks. http://www.satlive.org/satcompetition/2002/submittedbenchs.html.

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

[43] SAT 2002 Competition. http://www.satlive.org/satcompetition/.

[44] SAT 2003 benchmarks. `http://satlive.org/SATCompetition/2003/`.

[45] SAT 2003 challenge benchmark. `http://satlive.org/SATCompetition/2003/`.

[46] SAT 2003 Competition. `http://satlive.org/SATCompetition/2003/`.

[47] 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.

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

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

[50] 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.

[51] M. Swany and R. Wolski. Building performance topologies for computational grids. In *Proceedings of Los Alamos Computer Science Institute (LACSI) Symposium, 2002*, October 2002.

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

[53] TeraGrid. http://www.teragrid.org/.

[54] TeraGrid at Argonne National Laboratory. `http://www-unix.mcs.anl.gov/teragrid-anl/`.

[55] TeraGrid at Caltech Center for Advanced Computing Research. `http://www.cacr.caltech.edu/resources/teragrid/`.

[56] TeraGrid at Pittsburgh Supercomuting Center. `http://teragrid.psc.edu/`.

[57] TeraGrid at the National Center for Scientific Applications. `http://teragrid.ncsa.uiuc.edu/`.

[58] TeraGrid at the San Diego Supercomputing Center. `http://www.sdsc.edu/teragrid`.

[59] VGrADS. `http://hipersoft.cs.rice.edu/vgrads`.

[60] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running EveryWare on the computational grid. In *SC99 Conference on High-performance Computing Proceedings*, 1999.

[61] R. Wolski, J. Brevik, G. Obertelli, N. Spring, and A. Su. Writing programs that run everyware on the computational grid. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), 2001. available from `http://www.cs.ucsb.edu/~rich/publications/ev-results.ps.gz`.

[62] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 1999.

[63] R. Wolski, N. Spring, and J. Hayes. Predicting the cpu availability of time-shared unix systems on the computational grid. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing*, 1999.

[64] H. Zhang and M. Bonacina. Cumulating search in a distributed computing environment: A case study in parallel satisfiability, September 1994.

[65] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

[66] Y. Zhao, M. Moskewicz, C. Madigan, and S. Malik. Accelerating boolean satisfiability through application specific processing. In *Proceedings of the International Symposium on System Synthesis (ISSS), IEEE*, October 2001.