GridSAT: A Chaff-based Distributed SAT Solver for the Grid

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

Abstract

We present GridSAT, a parallel and complete satisfiability solver designed to solve non-trivial SAT problem instances using a large number of widely distributed and heterogeneous computational resources.

The GridSAT parallel algorithm (based on Chaff) uses intelligent backtracking, distributed and carefully scheduled sharing of learned clauses, and clause reduction. Our implementation for Computational Grid settings focuses on dynamic resource acquisition and release as a way of optimizing application execution. We show how the large number of computational resources that are available from a Grid can be managed effectively for the application by an automatic scheduler and effective implementation. GridSAT execution speed is compared against the best sequential solver as rated by the SAT2002 competition using a wide variety of problem instances. The results show that GridSAT delivers speed-up for all but one of the test problem instances that are of significant size. In addition, we describe how GridSAT has solved previously unsolved satisfiability problems and the domain science contribution these results make.

Keywords: Parallel, Distributed, Satisfiability, Computational Grid.

1 Introduction

The problem of determining propositional satisfiability (SAT) for arbitrary boolean expressions is an important problem in computer science and engineering. A typical SAT problem is formulated as the question "for a boolean expression, is there an assignment of boolean values to the variables of the expression that makes the expression evaluate as true?" In addition to playing a pivotal role in computability theory (Cook's original NP-completeness proof used SAT as its basis [10]), many engineering applications in the areas of circuit design [31], FPGA layout [25], Artificial Intelligence [21] scheduling [6] and Software verification [19] require solutions to arbitrarily complex SAT problems. In an industrial setting, computational SAT solvers are required to solve practical problems.

As a result, SAT solver development continues to be an active area of academic research and software development. There exists many sequential solvers [2, 16, 18, 5]. Most of these systems use "learning" – the formulation of additional deduced clauses during a solution run – to prune the search space of possible variable assignments. Learned clauses must be organized and stored so that the search heuristics can use them to prove various potential variable assignments infeasible (i.e. result in a logical conflict). Thus, solvers that employ clause learning form a local clause database that is heavily accessed (both read and updated) during a solution, and which can grow arbitrarily large.

Moreover, it is memory size and speed in addition to CPU availability that constrains the success of these sequential implementations. Typical solvers run until they have found a variable assignment that makes the original expression true, or until the learned clause database overflows the available memory. In the latter case, the solver cannot make any further progress and no information about the satisfiability of the given expression is produced.

Improvements in networking and CPU speeds in addition to the availability of clusters make it attractive to use multiple machines to solve SAT problems. There are many parallel SAT solvers [20, 32, 13]. However, current implementations do not use large sets of heterogeneous computational resources. Furthermore, there programming models do not address the additional requirements imposed by such an environment.

In this paper, we describe GridSAT – a new learning SAT solver implemented for Computational Grid execution – that dynamically acquires and releases resources during solution as a way of improving solver power. GridSAT is based on zChaff [2] which is the current world-record holder among all SAT solvers tested at the SAT2002 solver competition [28]. It uses zChaff as a solver core but in a new parallel form and it implements an efficient distributed clause database sub-

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

system that can acquire and release memory from a Grid resource pool on demand. Thus, while the core of the sequential computation is based on a previously defined implementation, GridSAT, using EveryWare, has been designed explicitly to manage the heterogeneity and dynamism inherent in Grid resource environments. It is, in this sense, an application developed from "first-principles" for the Grid and not a legacy parallel application that has been modified for Grid execution.

We describe the implementation of GridSAT using the EveryWare [35, 36] application development toolkit, and detail the resulting solver performance in terms of the SAT2002 challenge benchmark suite. Using a non-dedicated nationally distributed, and shared Grid, GridSAT outperforms zChaff (for the problems zChaff can solve) when it is run on the fastest processor available to our experiment in dedicated mode. In addition, GridSAT is able to solve problems from the benchmark suite that no other solver (including zChaff) has previously been able to solve, but for which analytical results are known. Finally, GridSAT has solved outright three previously unsolved problems in satisfiability for which neither analytical nor automatic techniques have previously proved successful. With these new results in the field of satisfiability, GridSAT is one of the first Grid programs developed from first principles to generate new domain science. Thus the contributions this paper makes can be summarized as

- an exposition of the GridSAT parallelization, scheduling, and implementation techniques that enabled successful Grid execution of a fine-grained storage-intensive application,
- an analysis of the performance comparison between GridSAT and the currently best-performing SAT solver known, and
- a discussion of the new domain science results that the GridSAT investigation has generated.

These results stem from two important innovations which, together, result in an automatic SAT solver that outperforms the best previously known solvers as measured by [28].

- **Scalable Distributed Learning**: We report on a method we have developed for distributed learning and sharing of automatically deduced clauses among a large set of hosts connected via a network. We believe that this methodology is effective enough to enable large, nationally distributed collections of machines to be used in parallel on individual problem instances.
- **Adaptive Resource Scheduling**: We describe an implementation of this technique for Computational Grid [14] execution environments. The goal of Grid systems is to permit resource intensive applications to dynamically acquire and release resources. These resources could be dedicated resources or they could belong to a globally available "pool" that is shared by all Grid users. We have developed an adaptive scheduling methodology that enables high-performance SAT solutions using shared Grid resources that are widely dispersed geographically.

As an empirical verification of these results, we compare our system to sequential zChaff [2] from which the core of our solver is derived. We use the SAT2002 [26, 27] benchmark suite as test applications, and a Computational Grid provided by our lab machines, the Grid Application Development Software (GrADS) project [3] and in some experiments Blue Horizon [7].

The rest of the paper is organized as follows. Section 2 presents the how SAT solvers work in general and how Chaff is implemented in particular. In section 3 we present our parallel version of the SAT algorithm. Section 3.3 describes the programming model we have used to implement the parallel solver in a Grid environment. We present a discussion of the experimental performance results in section 4. Finally we present related work and conclude in sections 5 and 6 respectively.

2 SAT Solvers and the Chaff Algorithm

A satisfiability problem is expressed as a boolean formula. 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*. 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.

The Chaff solver uses a basic solver algorithm. The performance of the algorithm is enhanced by using techniques for adding new deduced clauses. In this section we explain the basic algorithm and how conflicts are used to generate new clauses. We also present an example to illustrate how Chaff works. Chaff specific optimizations which require a larger problem to demonstrate are discussed at the end of this section.

2.1 The basic algorithm:

The basis of Chaff and many modern SAT solvers is the Davis-Putnam-Logeman-Loveland (DPLL) [12] 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 all possible variable assignments have been examined proving that the problem is unsatisfiable. 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 added as the algorithm proceeds.

After a new decision, the algorithm searches for *unit clauses*. A unit clause is a clause with only one literal (variable or its complement), without a determined truth value, 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 forcibly set 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 all 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. After each variable assignment, the algorithm inspects the clause database in search of unit clauses. This process is called *Boolean Constant Propagation* (BCP).

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 implication, a new decision is made and an additional decision level is added. In the latter case, a *conflict* happened. A conflict occurs when the implied variable is found to have been previously assigned the opposite value. When a conflict happens then it should be resolved. 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 requires 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 points to a level in the decision stack to which the algorithm can back-jump. Non-chronological back-jumping [40] occurs if the algorithm jumps by more than one decision level.

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 there are no more unknown variables to assign. However if the algorithm backtracks completely to the first decision level, there is a conflict due to deduced variable assignments at this decision level. Since assignments at this level are necessary for problem satisfiability then 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 [29, 22, 30] is the augmentation of the initial formula with additional implicates. The new clauses indicate search spaces which were found to have no solution because they result in conflicts. The presence of these clauses 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 implication 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 antecedent because they are not implied. In practice decision variables are given a fictitious clause. Initial and learned clauses are given indexes greater than 1, thus we use clause 0 (which does not exist) in this paper as antecedent for decision variables.

A *learned clause* is obtained by partitioning the implication graph into two sides. One partition called the *reason* side has 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 [40] in improving the algorithm's performance.

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 causes the solver to perform a non-chronological backtrack. After backtracking, the new decision level is the highest decision level among all the variables in the new learned clause. Chaff [2] 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. 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 point and the conflict site are on the conflict side. The next section presents an example to illustrate this method.

2.3 An Example SAT Problem Instance

The SAT formula for this example is shown at the top of Figure 1. It is made up of nine clauses and 14 variables. We start with an empty decision stack and with current decision level set to 0. First 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 and push another decision level. We choose (arbitrarily in this example) to set V_{10} to *false* 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 *false*, 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 this level we decide to set V_{11} to *true* making V_{11} the decision variable for level 6. In this case we have a cascading series of implications that lead to a conflict. The implication graph in Figure 1 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 use different cut rules from zChaff generating different learned clauses for this example. The subject of cut determination is an active research area among competing SAT solvers. 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 back track to the maximal decision level of all the decision variables involved in the conflict. This level is 4 which the decision level of $\sim V_9$. The new decision stack is also shown in Figure 1. Note that when using this method the new learned clause leads to an implication after backtracking involving the FirstUIP variable. In this implication the FirstUIP node V_5 is set to *false*.

2.4 Chaff Specific Optimizations

zChaff is an implementation of Chaff by *L. Zhang* from Princeton. There is another implementation *mChaff* [2, 24] which was independently developed by *M. Moskewicz*. Both versions implement the same optimizations which Chaff makes but differ in the implementation details. Chaff introduces two optimizations to the basic stack-based algorithm: a more efficient method for BCP and a new heuristic for choosing decision variables.

BCP accounts for a significant percentage (more than 90%) of execution time for most problems [2]. The goal of BCP is to find those clauses which result in an implication. An intuitive 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. To approximate this behavior Chaff marks two literals from each clause. When one of the marked literals becomes *false* because its variable has been assigned a value (i.e. *true* or *false*), then that literal is unmarked. If another literal with unknown value – that is not marked – from the same clause exists then it is marked. If no such literal exists then the clause is unit and the other marked literal is implied to be *true*. This technique reduces the number of times a clause is visited before it becomes a unit clause. Also it reduces the number of clauses visited after a variable assignment. The overall result is a more efficient BCP.

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 is found to have lower overhead compared to other heuristics.

3 GridSAT: SAT for the Grid

To build a Grid-enabled SAT solver using Chaff as the core algorithm, we need to address three significant challenges. GridSAT must

- parallelize the search algorithm that is navigating the space of possible truth assignments,
- distribute and share the clause database across Grid resources, and
- schedule the GridSAT application components dynamically so that they may take advantage of the best possible resources at the time and they can be used profitably by the algorithm.

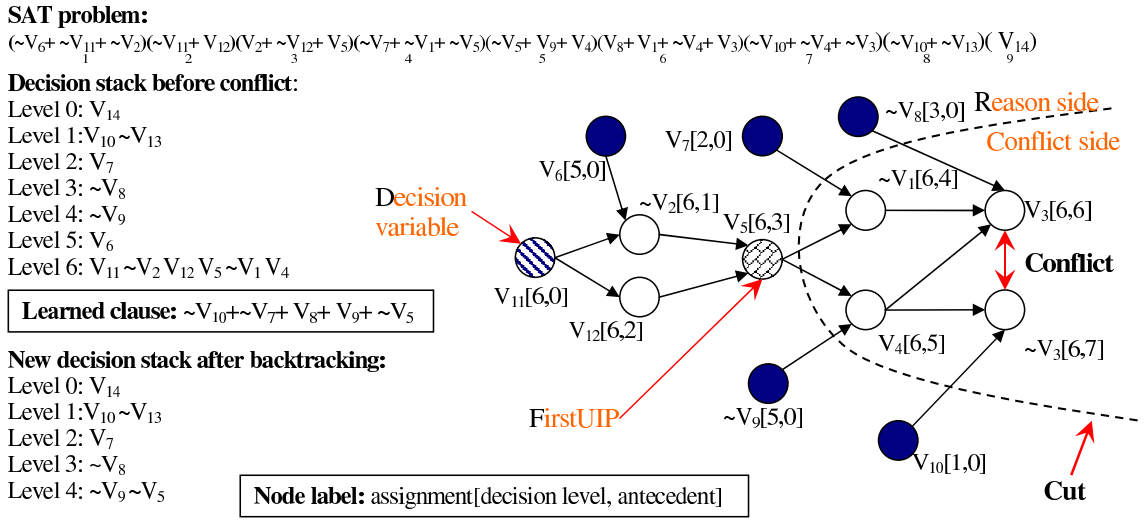

Figure 1: Example of conflict analysis with learning and non-chronological backtracking

SAT problems vary in terms of their resource requirements. Some problems are *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 should be able to use the resources according to the needs of the problem at hand.

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 database pertaining to each side. Finally, because the problem is so database intensive, the GridSAT scheduler attempts to add resources (machines and, more importantly, memory) only when the current resource set (which starts with one machine) becomes overloaded. In other words, the goal of the scheduler is to keep the execution as sequential as possible and to use parallelism only when it is needed.

3.1 Parallelizing SAT

For GridSAT the split process modifies the current problem and spawns a new one as shown in Figure 2. The old problem is modified by making all variables on the second decision level of the assignment stack part of the first decision level. The new problem generated consists of a set of variable assignments and a set of clauses. The variables assignments include all assignments from the first decision level and the complement of the first assignment in the second decision level. Thus insuring the splitting of the search space.

After splitting, each process (termed client in the GridSAT parlance) 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 level 0 of its decision stack as a result of the split. In the example in Figure, 2 client A can remove clauses 8 and 9 because $\sim V_{13}$ and V_{14} are *true* respectively. Client B can remove clauses 7, 9 and also the newly learned clause because of $\sim V_{10}$, V_{14} and $\sim V_{10}$ are *true* respectively. Because scarcity of memory is often the limiting factor, we also implemented this pruning optimization in the sequential version of zChaff that we use for comparison.

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 will be slower than if it were executed sequentially. This problem is occasionally referred to as the "ping-pong" effect [20].

3.2 Sharing and Distributing The Clause Database

Learned clauses from a client when shared with other clients can help prune a part of their search space. On the other hand, sharing clauses limits the kind of simplifications that can be made. For example, variables with known assignments (i.e. in first decision level) can be removed. However, removing them might make learned clauses only valid for the current client.

When new learned clauses are received from other clients, they are merged with the local clause database. These clauses will only be merged after the algorithm has backtracked to the first decision level. This simplification allows for a more

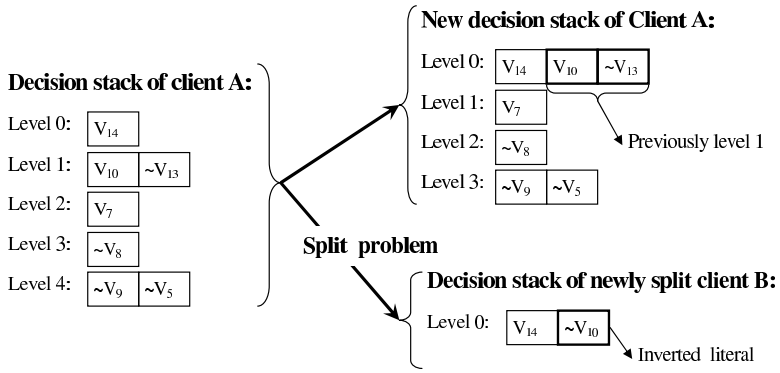

Figure 2: Example of stack transformation when a problem is split into two clients

straight forward implementation. It also insures that clauses are merged in batches. A learned clause can result in one of four cases:

- If the clause has only one unknown literal then it results in an implication.
- If the clause has more than one unknown literal then the clause is simply added to the set of learned clauses.
- If the clause has all literals *false* then there is a conflict and the subproblem is unsatisfiable.
- If the clause evaluates to *true* then the clause is discarded since it does not prune any part of the search space.

As described in [32], 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. Therefore GridSAT clients only share "short" clauses in order to minimize communication cost. Also 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). While we do not yet have a way of determining the length of the clauses to share automatically, GridSAT takes the maximum clause length as a parameter. As described in Section 4, the lengths we use in this investigation are 10 and 3.

3.3 GridSAT Scheduler Design and Application Implementation

GridSAT uses a master-client programming model. The execution starts at the master. The master is responsible for reading the problem file and generating the final output results. Only clients, but not the master, are assigned part of the search space to investigate. The master assumes three main functions:

- Resource Manager: Selects and monitors the set of available resources. Such information is culled from the Grid information system (e.g. Globus MDS [11], the NWS [37, 33, 38], etc.).
- Client Manager: Monitors the state of each client. It also allows more clients to join at runtime. These clients can be a result of a batch job reaching the head of its queue or other additional resources becoming available.
- Scheduler: maps a given subproblem to the processor it deems best to add to the application's resource pool.

After initialization the master queries for the list of available resources and launches an empty client on each resource. If a resource is a batch controlled system, it simply submits a job request. When a clients starts successfully it contacts the master and registers with it. The master ranks the set of registered clients based on the resources they are running on. The rank of a computational resource is defined according to its processing power and memory capacity as forecast by the Network Weather Service [34, 37]. GridSAT uses static information and application instrumentation to determine these performance measures when it is not configured to use NWS or Globus MDS information. In this case each client queries the system for the size of free memory and will only use up to 60% of it. This strategy is implemented to avoid running the system of of memory even when other less memory intensive applications are later launched by other users. In this manner the application evades being killed by some Operating Systems[1]. On some systems, it is only possible to query for the size of total memory and not the size of free memory. In this case the maximum memory a client can use is provided as an argument. GridSAT clients will terminate if the initial free memory size is below a given minimum (currently set to 128 MBytes). Clients which run on low memory resources are less likely to solve a given subproblem. In addition, they will experience memory shortages quickly and will tend to split more frequently causing a lot of communication overhead.

When a client starts it does not have a problem to solve so we call it "idle". GridSAT uses a limited form of recovery.

[1]Linux kernel implements a memory-shortfall policy that is more likely to kill a large-memory job when the machine runs out of memory.

When an idle client is killed for any reason, the master becomes aware of it and marks the resource as "free". In case the master needs more resources, it tries to restart clients on free resources.

The first client to register with the master is sent the entire problem to solve. Each client monitors its own memory and CPU usage and when it predicts that its resources are about to be exhausted it notifies the master. Thus, the decision to add a resource is made locally by a client that has over-run its resource limits and not by a central scheduler. Upon receipt of a notification from a client that a problem split is required, the master searches within the resource pool for the highest ranked idle resource.

Once a resource is selected, an empty client is initiated on the machine if needed. The client running on the selected resource then contacts the original client requesting a split and communicates with it directly to divide the problem and obtain relevant shared clauses with which to initially populate its database. Clients can communicate through the master. However the use of direct client-to-client communication is more efficient especially for the messages sent during split which are very large (up to 100s of MBytes in size).

A client decides to split its problem when it "believes" it will soon run out of memory, or when it deems the problem it is working on solving big enough to warrant splitting. The former decision, as previously discussed, is based on available machine memory. If a memory shortfall is predicted (because the memory loads have changed or the clause database is growing) the client requests a split. Determining the "difficulty" of a particular subproblem, however, is challenging since it is problem specific. In the current version of GridSAT, the client uses a simple time out heuristic based on the notion that a subproblem having run for a "long" time is a subproblem that probably benefit from added resource.

A client records the time it required to send or receive a problem. When twice this time period expires, the client requests more resource from the master to help solve the current subproblem on the assumption that a long running problem will continue to be a long running problem. This time out period is used to offset communication cost so that clients are not busy splitting a problem instead of doing useful work. This cost becomes more significant because of the large number of possible clients and communication overhead. An easy subproblem is better handled by one machine. The harder it becomes (represented in GridSAT by the length of time it runs) the more resources are needed. Generated subproblems (i.e. assignment stack and clauses) are large and require a significant period of time to send over a wide area network. Thus, the scheduler must attempt to balance the benefit of extra processing power against the expense of communicating the necessary state.

An example communication scenario is shown is Figure 3. The figure shows the communication messages involved in splitting the subproblem assigned to client A. The figure shows five messages. In (1) client A notifies the master that it wishes to split its subproblem. The master replies by message (2) indicating which of client A's peers, client B in this case, is available to split the problem with client A. At this stage client A communicates directly with client B using message (3) which contains the new subproblem. Using Peer-to-peer communication to send this message enhances the overall performance since this is by far the largest message sent. This message varies in size from 10 KBytes to 500 MBytes, but is 100s of MBytes in size on average. Using messages (4) and (5) clients B and A respectively notify the master of the success or failure of their communications.

3.4 Work Backlog

For some long running instances all available resources might become busy. At the same time, clients may request the master to split their subproblems. The master records these requests and keeps 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 which 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. In addition, the master can direct a client to migrate the current problem instead of splitting it. This migration is useful when a problem is migrated from a single remote resource to a set of well-connected more powerful idle nodes. For example jobs from a single machine are migrated to a cluster of machines when the cluster becomes free. Because clients communicate as peers when a split occurs, moving a "hard" subproblem from a single remote resource to a field of idle resources allows the master to select machines that are near the splitting client. After migration the scheduler can direct the client to split with near-by clients leading to more efficient use of the available bandwidth resource.

Finally, the master terminates in one of three cases:

- All the clients are idle which means that the instance is unsatisfiable.
- one of the clients finds a satisfiable solution. This client sends the assignment stack to the master which verifies that the stack satisfies the problem. This procedure is accomplished in order of the size of the problem.
- An error occurs when the master times out or a client runs out of memory and is killed by the operating system.

For a fair comparison between the performance of GridSAT and zChaff we assume that there are no resource failures during the experiments. As mentioned earlier, GridSAT implements a limited form of resilience in the presence of failures. The current implementation, however, will not tolerate a machine crash or a Linux "out-of-memory killer" process termination for clients which are working on a subproblem.

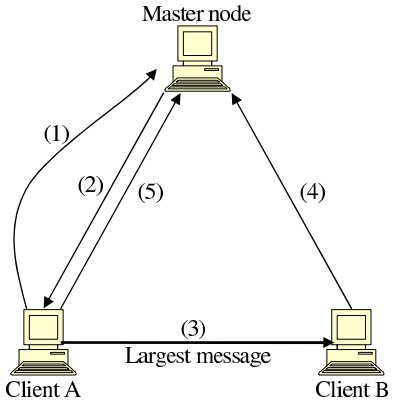

Figure 3: Communication scenario of splitting the subproblem assigned to client A with client B

In future versions of GridSAT, we will provide a better way for the program to continue in the presence of resource failure. We can use two types of checkpoints:

- Light checkpoint: Only checkpoint level 0 for each client. In this case checkpoints for a client will be updated only when more variables are added to decision level 0.
- Heavy checkpoint: In addition to the light checkpoint, we add 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 at regular time intervals in addition to the instances when level 0 is augmented.

In both cases, the initial set of clauses are obtained from the problem file. The light checkpoint is necessary to record progress but does not require a lot of disk space. However check-pointing learned clauses requires a lot space (about .5 Gigabytes per client). The heavy checkpoint method would require a system capable of handling large checkpoints. We have implemented an experimental prototype of GridSAT which supports a comprehensive form of check-pointing and fault-tolerance which we have not analyzed yet. We will report on this in the future.

4 Experimental Apparatus and Results

To investigate the efficacy of GridSAT, we combine resources from the GrADS [3] testbed grid with additional UCSB machines and (possibly) nodes culled from the IBM Blue Horizon [7] located at the San Diego Supercomputer Center (SDSC). The **Gr**id **A**pplication **D**evelopment **S**oftware (GrADS) project [3, 17] 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. The baseline Grid infrastructure is provided by Globus and the NWS, upon which is layered a set of programming abstractions collectively termed *GrADSoft*. GrADSAT [9] (note the "A" in the spelling) is an early prototype SAT solver we developed using GrADSoft tools. In this work we extend GrADSAT to use other resources (such as non-GrADS machines at UCSB and the IBM Blue Horizon) that do not currently benefit from these sophisticated Grid programming tools. GridSAT components (i.e. master and client) use the EveryWare [35, 36] messaging system for communication.

During our experiments, none of the resources we used were dedicated to our use. The GrADS testbed, the UCSB machines, and the Blue Horizon 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. We consider this to be a realistic scenario for Computational Grid computing, but it makes repeatable timings of similar problems (particularly those that run for long periods) difficult. In particular, the Blue Horizon is a batch controlled system where 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 GrADS 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.

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 [26]. 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* [27]. The solvable category contains problem instances that SAT solvers have been known to solve correctly. They are useful for comparing the speed of compet-

itive 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 which 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.

We selected examples from both categories in order to have the widest possible instance variety and so as not to bias our results unduly in favor of "easy" problems. The total number of instances we investigate is 42 divided as composed

- 22 instances from the solvable industrial benchmark category,
- 15 instances from the solvable hand-made/random benchmark category,
- 11 instances from the challenging benchmark category,

In the first set of 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.

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 which would have maximal effect without increasing significantly the overhead of clause sharing. Also the time out for clients to request that their problems be partitioned is set to 100 seconds. 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. However, in an effort to complete this submission, we noted that for all of the cases where zChaff terminated, less than 12000 seconds were used. We then chose 12000 seconds for the challenging set. 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.

In the second set of experiments we used 27 machines an additional machine as a master node, and nodes from the IBM Blue Horizon (once they became available). The machines are distributed among three sites: one cluster of 16 nodes at UIUC, 3 desktop machines at UCSD and 8 desktop machines at the University of California, Santa Barbara (UCSB). Only the UIUC and UCSD machines are part of the GrADS testbed. The machines had a variety of hardware and software configurations.

In these experiments the maximum size of learned clauses shared is 3. During these experiments we started immediately running on the set of machines described previously. We also submitted a request to the Blue Horizon scheduler for 100 node job lasting twelve hours. Each Blue Horizon node had 8 CPUs per node and 4 gigabytes of memory per node. The wait period for such a job on the average is about 33 hours. If a problem was not solved by the end of the 12-hour Blue Horizon job, the whole GridSAT run terminated. If the problem was solved before the batch job ran then GridSAT will terminate and the job queued from the Blue Horizon is canceled.

4.1 Results

We compare the solvers by comparing the time it takes to solve a SAT problem. Since performance is important we disabled the instrumentations in the application. Therefore this section does not contain resource performance metrics as instrumentation influences performance negatively. In [35], it mentions that instrumentation reduces performance by as much as 50%.

The first set of results are presented in Table 1. 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 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.

The problem instances in Table 1 are split into three categories. The first section 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.

The second category of problems represent those 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 [28]. 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.

The rest of the seven instances in this second category 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.

The final set of input files represent the SAT problems which were not solved by either GridSAT nor zChaff. In the second set of experiments shown in Table 2 we removed the slower, less-well provisioned machines (250 MHz Pentium IIs with 128 megabytes of memory each at UIUC) from consideration and added the possibility of running on the IBM Blue Horizon. At the time of application initiation, a batch job requesting 100 nodes from the Blue Horizon was submitted. If and when the nodes were allocated, the clients running there would contact the application master and request work. Until that time, however, the application would run using the nodes available to it from the GrADS testbed and UCSB thereby covering the start-up latency (approximately 33 hours on average) associated with the Blue Horizon.

Because we avoided allocating work to slower resources, both instances *rand-net70-25-5.cnf* and *glassybp-v399-s499089820.cnf* were solved before the Blue Horizon job started causing GridSAT to cancel the job request. The *par32-1-c.cnf* problem ran for 33 hours on the GrADS testbed and UCSB machines before the Blue Horizon nodes were allocated and an additional 8 hours on the Blue Horizon before GridSAT solved it. To determine the Blue Horizon savings that the interactive grid resources bring, we re-launched the application on the Blue Horizon alone. After waiting again for approximately 33 hours in queue, the 300 Blue Horizon processors solved the problem instance in approximately 12 hours. Thus, the non-dedicated, nationally distributed Grid time saved approximately $(12-8)(hours)*8(cpus/node)*100(nodes) = 3200$ processor hours of Blue Horizon time while shortening the time to solution by 4 hours.

4.2 Discussion

The results show that the parallel solver is more efficient than a sequential one. This may be explained as the combination of the following reasons:

- More resources – specifically CPUs – can cover more of the search space during the same time than a single CPU would be able to.
- The splitting process and removal of inconsequential clauses results in smaller subproblems.
- As the algorithm progresses, a lot of memory is devoted to storing antecedent clauses. Once a clause becomes an antecedent it is only used for backtracking. It is the clauses which can lead to new implications that are valuable to the current search step. A sequential solver cannot delete antecedent clauses and might have no memory space to store new clauses. This memory shortage causes new learned clauses to be deleted shortly as their space is used to store more recently learned clauses. The result is a degradation of the algorithms performance. In a parallel solver a client that runs into this problem might be relieved when it splits. This happens because unnecessary clauses will be discarded and therefore more memory will be available.

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. PSATO [39] 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 [20] which is the parallel implementation of SATZ [23]. 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 [32] implements a different algorithm for clause sharing. In addition,

File name	SAT/UNSAT/ UNKNOWN	zChaff (sec)	GridSAT (sec)	Speed-Up	Max # of clients
Problem solved by zChaff and GridSAT					
6pipe.cnf	UNSAT	6322	4877	1.23	34
avg-checker-5-34.cnf	UNSAT	1222	1107	1.10	9
bart15.cnf	SAT	5507	673	8.18	34
cache_05.cnf	SAT	1730	1565	1.11	34
cnt09.cnf	SAT	3651	1610	2.27	12
dp12s12.cnf	SAT	10587	532	19.90	8
homer11.cnf	UNSAT	2545	1794	1.42	10
homer12.cnf	UNSAT	14250	4400	3.24	33
ip38.cnf	UNSAT	4794	1278	3.75	11
rand_net50-60-5.cnf	UNSAT	16242	1725	9.42	20
vda_gr_rcs_w8.cnf	SAT	1427	681	2.10	15
w08_14.cnf	SAT	14449	1906	7.58	34
w10_75.cnf	SAT	506	252	2.01	2
Urquhart-s3-b1.cnf	UNSAT	529	526	1.01	4
ezfact48_5.cnf	UNSAT	127	196	0.65	1
glassy-sat-sel_N210_n.cnf	SAT	7	68	0.10	1
grid_10_20.cnf	UNSAT	967	3165	0.31	12
hanoi5.cnf	SAT	2961	1852	1.60	33
hanoi6_fast.cnf	SAT	1116	831	1.34	4
lisa20_1_a.cnf	SAT	181	243	0.75	2
lisa21_3_a.cnf	SAT	1792	337	5.32	4
pyhala-braun-sat-30-4-02.cnf	SAT	18	84	0.21	1
qg2-8.cnf	SAT	180	224	0.80	2
Problems solved by GridSAT only					
7pipe_bug.cnf	SAT	TIME_OUT	5058	–	34
dp10u09.cnf	UNSAT	TIME_OUT	2566	–	26
rand_net40-60-10.cnf	UNSAT	TIME_OUT	1690	–	30
f2clk_40.cnf	UNSAT(*)	TIME_OUT	3304	–	23
Mat26.cnf	UNSAT	MEM_OUT	1886	–	21
7pipe.cnf	UNSAT	MEM_OUT	6673	–	34
comb2.cnf	UNSAT(*)	MEM_OUT	9951	–	34
pyhala-braun-unsat-40-4-01.cnf	UNSAT	MEM_OUT	2425	–	34
pyhala-braun-unsat-40-4-02.cnf	UNSAT	MEM_OUT	2564	–	34
w08_15.cnf	SAT(*)	MEM_OUT	3141	–	34
Remaining problems					
comb1.cnf	*	TIME_OUT	TIME_OUT	–	34
par32-1-c.cnf	SAT	TIME_OUT	TIME_OUT	–	34
rand_net70-25-5.cnf	UNSAT	TIME_OUT	TIME_OUT	–	34
sha1.cnf	SAT	TIME_OUT	TIME_OUT	–	34
3bitadd_31.cnf	UNSAT	TIME_OUT	TIME_OUT	–	34
cnt10.cnf	SAT	TIME_OUT	TIME_OUT	–	34
glassybp-v399-s499089820.cnf	SAT	TIME_OUT	TIME_OUT	–	34
hgen3-v300-s1766565160.cnf	*	TIME_OUT	TIME_OUT	–	34
hanoi6.cnf	SAT	TIME_OUT	TIME_OUT	–	34

(*): problem solution is unknown

Table 1: GridSAT and zChaff SAT2002 Benchmark Results on GrADS testbed

File name	SAT/UNSAT/UNKOWN	GridSAT(sec)
comb1.cnf	*	X
par32-1-c.cnf	SAT	33hrs+(8hrs on BH)
rand-net70-25-5.cnf	UNSAT	30837
sha1.cnf	SAT	X
3bitadd-31.cnf	UNSAT	X
cnt10.cnf	SAT	X
glassybp-v399-s499089820.cnf	SAT	5472
hgen3-v300-s1766565160.cnf	*	X
hanoi.cnf	SAT	X

(*): problem solution is unknown

Table 2: GridSAT results using testbed and Blue Horizon on the harder problems. All these problems were not solved by zChaff or any other solver before.

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 [13]. 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 [41]. 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-slave applications [4, 8, 1, 15]. NetSolve [8] 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. The satisfiability problem is different from most existing applications because it does not have a predictable runtime or resource needs .

6 Conclusion

The paper presents a new application which would benefit when run in a Grid environment. This application is a SAT solver. We implement – GridSAT – a satisfiability solver which runs on a set of widely distributed commodity computational resources. The distributed solver dynamically acquires and releases computational nodes. In addition, new clients can join dynamically. The solver is adaptive to the problem's resource needs. Easy problems which take shorter time to solve use a small number of resources. Other harder instances which take longer to solve use more resources.

The experimental results show that a variable amount of speed-up is obtained compared to a highly optimized sequential solver over a wide range of instances. The more significant result was that GridSAT solved harder instances. Some of these instances were not solved before. We conclude that the use of dynamic resource acquisition and release coupled with an effective, first-principles Grid implementation yields a faster time to solution on problems that have known solutions, and will continue to generate new satisfiability results for academic and industrial researchers.

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

[3] 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".

[4] 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.

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

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

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

[8] 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.

[9] W. Chrabakh and R. Wolski. GrADSAT: A Parallel SAT Solver for the Grid. Technical Report 2003-05, UCSB, March 2003.
[10] S. A. Cook. The complexity of theorem-proving procedures. *Third Annual ACM Symposium on Theory of Computing*, 1971.
[11] 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.
[12] M. Davis, G. Logeman, and D. Loveland. A machine program for theory proving. Communications of the ACM, 1962.
[13] S. L. Forman and A. M. Segre. Nagsat: A randomized, complete, parallel solver for 3-sat. SAT2002, 2002.
[14] I. Foster and C. Kesselman, editors. *The Grid – Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.
[15] G. Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, University of California, San Diego, May 2001.
[16] 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.
[17] GrADS. `http://hipersoft.cs.rice.edu/grads`.
[18] 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.
[19] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. *International Symposium on Software Testing and Analysis*, 2000.
[20] 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.
[21] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 359–379, August 1992.
[22] T. Larrabee. Efficient generation of test patterns using boolean difference. pages 795–802.
[23] C. M. LI. A constrained-based approach to narrow search trees for satisfiability. Information processing letters 71, 1999.
[24] M. Moskewicz. http://www.ee.princeton.edu/ chaff/index1.html.
[25] 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.
[26] SAT 2002 benchmarks. http://www.satlive.org/satcompetition/2002/submittedbenchs.html.
[27] SAT 2002 challenge benchmark. http://www.ececs.uc.edu/sat2002/sat2002-challenges.tar.gz.
[28] SAT 2002 Competition. http://www.satlive.org/satcompetition/.
[29] 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.
[30] J. M. Silva and K. Sakallah. Grasp - a new search algorithm for satisfiability. ICCAD. IEEE Computer Society Press, 1996.
[31] J. P. M. Silva. Search Algorithms for Satisfiability Problems in Combinational Switching Circuits. Ph.D. Thesis, The University of Michigan, 1995.
[32] 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.
[33] M. Swany and R. Wolski. Building performance topologies for computational grids. In *Proceedings of Los Alamos Computer Science Institute (LACSI) Symposium, 2002*, October 2002.
[34] R. Wolski. Experiences with predicting resource performance on-line in computational grid settings. *ACM SIGMETRICS Performance Evaluation Review*, 30(4), March 2003.
[35] 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.
[36] 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`.
[37] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 1999.
[38] 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.
[39] H. Zhang and M. Bonacina. Cumulating search in a distributed computing environment: A case study in parallel satisfiability, September 1994.
[40] 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.
[41] 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.