GrADSAT: A Parallel SAT Solver for the Grid

UCSB Computer Science Technical Report Number 2003-05

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

Abstract

We present GrADSAT, a parallel satisfiability solver aimed at solving hard SAT instances using a large number of widely distributed commodity computational resources. The GrADSAT parallel algorithm uses intelligent backtracking, sharing of learned clauses and clause reduction. The distributed implementation allows for dynamic resource acquisition. We show how the large number of computational resources and communication overhead influence the implementation strategy. GrADSAT is compared against the best sequential solver using a wide variety of problem instances. The results show that GrADSAT delivers speed-up on most instances. Furthermore it is capable of solving problem instance which were never solved before.

Keywords: Parallel, Distributed, Satisfiabilty, Computational Grid.

1 Introduction:

Propositional satisfiability (SAT) is an important problem in computer science from a theoretical perspective. It is also pivotal for a wide range of practical applications. Such applications include circuit design [18], FPGA layout [13], Artificial Intelligence [11] and scheduling [4]. SAT solvers represent a powerful tool for solving problem instances for these applications. In particular, SAT solvers are used in the verification of circuit designs resulting in accelerated development of new circuits.

Initial work to develop automatic SAT solving programs primarily focuses on the Davis-Putname-Longemann-Loveland (DPLL) algorithm [5] for solving SAT instances. Solutions to practical problems, however, remained difficult to achieve. More recent research [1,3,8,9] has resulted in algorithmic and implementation advances that make it possible to solve problem instances generated by practical applications. Because of the importance of SAT results to the engineering community, a suite of benchmark [14] problems (some with known solutions some not) has been developed to test the efficacy of solver programs and an annual competition [16] is held in which different solver implementations are compared head-to-head.

Even with this interest and the significant advances that have been achieved, there are still many problems considered out of reach for the currently available solvers. Some of these problems are artificially generated, but most of the so called "hard" problems are drawn from real applications. For example, simple planning problems from the field of artificial

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

intelligence often yield large SAT instances. Moreover, in the field of circuit design, circuits are typically verified piecewise, as separate modules, to avoid the larger problem instance generated by a complete circuit. This type of verification is often error prone since it fails to capture the interactions between separately verified circuits that a complete verification would test.

There are two significant impediments to the realization of higher-quality SAT solvers. Without a proof that P-space is equal to NP-space, the best that can be hoped for are new heuristics for navigating the search space of possible variable assignments as efficiently as possible. Even with good heuristics, many of the problems that can now be solved require large amounts of computational cycles to complete. There have been several efforts to parallelize solver heuristics as a way of obtaining a faster time to solution, but most of the leading heuristics available at present maintain a carefully structured database of partial solutions during the search process. Updating and maintaining this database is generally termed "learning" and is data intensive. Some parallel solvers [10, 21] employ sophisticated learning techniques. It is generally too costly (in terms of communication overhead) for processors working on different parts of the search space to share information. Alternatively, in the case of PaSAT [19] (a recently developed parallel solver that learns), scalability is limited to a small number of tightly coupled processors.

Moreover, because the execution time for SAT solvers depends, in large part, on the problem being solved (i.e. some problems are solved quickly, some not at all) it is difficult to partition the search space among the available processors in a way that results in a significantly faster time to solution. For these reasons, sequential SAT solvers are more commonly used.

In this paper, we describe two important innovations which, together, result in an automatic SAT solver that outperforms the best previously known solvers as measured by [16].

- **Scalable Distributed Learning**: We report on a method we have developed for distributed learning and sharing of automatically deduced clauses amongst 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 [7] execution environments. The goal of Grid systems is to permit resource intensive applications to dynamically acquire and release resources from 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 [1] from which the core of our solver is derived. We use the SAT2002 [14, 15] benchmark suite as test applications, and a Computational Grid provided by the Grid Application Development Software (GrADS) project [2]. Using machines located at the various institutions participating in GrADS (which are located throughout the United States) our system — *GrADSAT* — out performs sequential zChaff using the best machine available to our experiment. In addition, GrADSAT has been able to solve previously unsolved "hard" problems from the benchmark suite. As such, we believe that our efforts constitute a significant advance in SAT solver capability and does so using a scalable and widely distributed computing paradigm.

The rest of the paper is organized as follows. In Section 2 we present our parallel SAT algorithm. Section 3 describes the programming model we have used to implement for Computational Grid settings. Section 4 presents our experimental setup and the results we have observed, in Section 5 we discuss related work, and Section 6 describes our concludes.

2 zChaff and GrADSAT

The core of our satisfiability solver is based on zChaff [1] developed by Sharad Malik's group at Princeton University. The zChaff solver is a "complete" solver. As such, it is guaranteed to find an instance of satisfiability if the problem is satisfiable, or to terminate proving that the problem is unsatisfiable. The general satisfiability problem is expressed to zChaff as a set of boolean *variables*. Each instance of a variable or its complement is termed a *literal*. A *clause* is

```
while(1) {
    //GrADSAT insertions: merge learned clauses, split problem
    //pick the next variable to assign heuristically
    if(decide_next_branch()) {
        //find implications and look for conflicts
        while(deduce()==conflict) {
            //find backtrack level and learn from conflict
            blevel = analyze_conflicts();
            if(blevel==0) // backtracked to top level
                return UNSATISFIABLE;
            else // unset variable assignments in reverse order till blevel
                back_track(blevel);
        }
    }else
        return SATISFIABLE;
}
```

Figure 1. DPLL Algorithm with learning and backtracking

an injunction of literals. Finally, a SAT problem is expressed as a conjunction of clauses. This problem decomposition corresponds to Conjunctive Normal Form (CNF) and zChaff takes, as input, a standard ASCII text representation of CNF.

The zChaff solver itself implements the DPLL algorithm with learning (pseudocode for which is shown in Figure 1).

As the algorithm proceeds it builds a data structure called a *decision stack*. The first decision level contains variable assignments necessary for the problem instance to be satisfiable. The DPLL algorithm consists of a decision heuristic for picking a variable that does not have an assigned value and speculatively assigning a value to that variable. This value assignment is propagated through all clauses that contain the variable. The system iteratively assigns a value to a variable in a clause, and then checks to see if the clause has been satisfied. If the assignment results in a clause that evaluates to *false*, a conflict has been reached and propagation terminates. At this point in the algorithm, the particular variable assignment is invalid since it results in a clause that is false. That is, the algorithm has "learned" that a particular variable assignment cannot be made if the overall expression is satisfiable. The conflict is analyzed as described in [17] resulting in a new clause that reflects this learned information, and the algorithm backtracks to the appropriate level. Eventually the algorithm terminates under one of two possible conditions. In case the problem is satisfiable a set of variable assignments where all clauses evaluate to *true* is found. However if the problem is unsatisfiable then the algorithm will backtrack completely to the first decision level.

2.1 Partitioning the Problem Space

To apply a parallel search technique to SAT, the initial problem is typically split into subproblems, 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.

GrADSAT follows this model by splitting the work assigned to a processor preceding a new decision as marked in figure 1. 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.

In order to alleviate memory usage inconsequential clauses are removed. The set of clauses in both problems include all clauses which do not evaluate to *true* because of the associated assignment stack. Because scarcity of memory is often the

limiting factor, we also implemented this pruning optimization in the sequential version of zChaff 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 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 [10].

2.2 Sharing Learned Clauses

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.

When new w learned clauses are received from other clients, they are merged before choosing the next variable assignment as indicated in figure 1. These clause will only be merged after the algorithm has backtracked with only one decision level on decision stack. This allows for simpler 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 that 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 [19], 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 GrADSAT 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. While we do not yet have a way of determining the length of the clauses to share automatically, GrADSAT takes the maximum clause length as a parameter. As described in Section 4, the length we use in this investigation is 10.

3 GrADSAT Implementation

Networks today are faster, more reliable and pervasive. These networks connect a large set of heterogeneous computational resources, not all of which belong to any single machine. The purpose of Computational Grids [7] is to enable applications to take advantage of these resources as an ensemble. The Grid Application Development Software [2] is investigating techniques that facilitate Grid application development in scientific and engineering fields.

3.1 Programming Model

GrADSAT uses a master-slave programming model. The execution starts at the master. The master is responsible for reading problem file and generating the final output results. The master also manages the clients. This activity consists of starting each client, monitoring their state, and coordinating problem execution. The master ranks possible target execution sites according to processing power and memory capacity. Such information is obtained from the GrADS information server.

The first client is automatically started since we need at least one client to solve any problem. When the client's resources are about to be exhausted it notifies the master. Upon receipt of a notification the master searches within the resource pool

for the highest ranked idle resource. A new client is initiated and it communicates with the spawning client in order to split the original problem. While executing, a client monitors its memory usage and the amount of time it has spent on a particular subproblem. If a client uses more than a threshold percentage of memory even after deleting all unnecessary clauses, it is then risking running out of memory and thus will eventually stop. Therefore, its problem must be split.

A time out period is given every client. When this time period expires, the client requests more resource from the master to help solve the current sub-problem on the assumption that a long running problem will continue to be a long running problem. This time out period is chosen to offset communication cost. The time period is chosen to be large enough so that clients are not busy splitting a problem instead of doing useful work. This problem becomes more significant because of the large number of clients and communication overhead. An easy problem is better handled by one machine. The harder it becomes (represented in GrADSAT 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. Using the GrADSoft tools, the scheduler can determine how fast each machine is, how much memory is available, and the performance of the network connectivity between machines. It uses this information to determine which resource to acquire once a decision to split is made.

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 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.

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, or
- an error occurs when the master times out or a client runs out of memory.

In future versions of GrADSAT, we will provide a way for the program to continue in the presence of resource failure. The current implementation, however, will not tolerate a machine crash or process crash.

4 Experimental Setup and Results

The experimental results were obtained by running a set of test problems on the GrADS testbed. The GrADS testbed was not dedicated to running GrADSAT, but rather was being used by various GrADS researchers at the time of the experiment. As such, other applications might have 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. Resources are federated to the Grid by their owners who maintain local control. They are under no obligation to announce their resource usage, nor must they warn other users that a resource will be reclaimed entirely. If it were possible to dedicate all of the GrADS resources to GrADSAT, 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.

The suite of files composing our benchmark were selected from the set of benchmarks submitted to the SAT2002 conference [14]. These benchmarks are used to rate all competing solvers. They include industrial and hand-made or randomly generated instances. We selected challenging examples of each type in order to have the widest possible instance variety and so as not to bias our results unduly in favor of “easy” problems. We also selected a number of instances from the problem set designated as *challenging* [15]. These are problems which were deemed hard by all solvers in the 2002 competition. Some of the challenging instances come from the industrial and handmade benchmarks. The total number of instances we investigate is 42 divided as composed

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

- 11 instances from the challenging benchmark category,

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 amongst 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. The timeout was set to a total of 6000 seconds for GrADSAT when running the industrial and handmade benchmarks. For the challenging benchmarks, we double the overall time out to 12000 seconds. In the case of sequential zChaff we use 18000 seconds for the industrial and handmade cases. 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 GrADSAT.

4.1 Results

The final results are presented in Table 1. The second column represents the instance solution. A question mark (?) means that the solution was unknown before we attempted to solve it with GrADSAT.

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 GrADSAT.

The problem instances in Table 1 are split into three categories. The first category represents the set of instances which were solved by both zChaff and GrADSAT. On the small instances (ones that complete relatively quickly) where communication costs are significant we notice that zChaff running on a single machine outperforms GrADSAT. The slowdown however is not very significant because the actual time is short. For instances with long running times GrADSAT shows a wide range of speed-ups ranging from almost none to almost 20 for *dp12s12*. Because GrADSAT was using more machines it was capable of covering much more of the search space even when the runtimes were comparable. In only one relatively long running instance, *grid_10_20*, did GrADSAT show a slowdown. The maximum number of active clients for the entire problem only reached twelve during its execution. With little sharing, parallelism did not seem to improve performance. This problem comes from a non-realizable circuit design.

The second category of files represent the set of files that GrADSAT was able to solve while zChaff either timed-out or ran out of memory. In addition, only three out of the ten files in this category were solved by another solver during the SAT2002 competition [16]. Note that zChaff was crowed 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. Moreover, each of these problems was solved by only one other complete solver.

The rest of the seven instances in this second category have only solved by GrADSAT to the best of our knowledge. GrADSAT was able to solve harder instances. Three of the solved instances were part of the challenging benchmark for which results were originally unknown. The other four had known answers, but no automatic generalized solver had been able to correctly generate them.

The final set of input files represent the SAT problems which were not solved by neither GrADSAT nor zChaff. These

File name	SAT/UNSAT/ UNKNOWN	zChaff (sec)	GrADSAT (sec)	Speed-Up	Max # of clients
Problem solved by zChaff and GrADSAT					
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_15.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 GrADSAT 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. GrADSAT and zChaff SAT2002 Benchmark Results on GrADS testbed

problems might require more resources or longer time to solve.

4.2 Discussion

We find these results somewhat startling. At the inception of this effort, we did not expect that GrADSAT would be able to solve unknown problems using only 34 processors from the GrADS testbed. Compared to various national-scale Grid efforts such as TeraGrid [20], the maximum processor count is relatively meager. Additionally, the purpose of the testbed is to support Grid application and software tool development — not to achieve new scientific results. As such, many of the machines are considered obsolete by current standards (one of the UIUC clusters consists of 266 MHZ, Pentium-II machines with 128 megabytes of RAM each, for example). Moreover, other GrADS researchers were using the machines during our experiment. We are fairly certain that no other large-scale application tests interfered with our timings, but normal development (compilation, small test runs, editing, etc.) continued. Indeed, the UCSD machines are student and staff desktop workstations that support the day-to-day activities of their primary users.

For these reasons, our original assumption was that a Grid-enabled SAT solver executing in this environment would solve known problems faster, but would not necessarily add solution power to that provided by zChaff. The combination of effective splitting and sharing with the ability to acquire and release resources dynamically enables our results.

In that vein, the final column in Table 1 bears closer study. The number of active clients varies across different problem instances. As mentioned previously, GrADSAT starts with one client. It then "grows" out onto the Grid from its launching point when the scheduler determines that more resources will speed execution. More resources become active when a problem is partitioned into two subproblems. Partitioning (splitting) is triggered by a time out (the subproblem ran too long) or because of a memory shortfall. When a new resource is to be acquired, the scheduler consults the GrADS Information System to determine the most suitable machine to add based on processor speed and available memory. After a subproblem completes, the machine it was running on is released back into the Grid. The GrADS infrastructure provide up-to-the-minute estimates of future performance from all machines. Thus the "best" machine to add varies from moment to moment due to contention by other applications. GrADSAT makes the decision about when to split and what machine to acquire at each split automatically, based on a constant re-evaluation of likely future performance benefit.

5 Related Work:

There are several parallel solvers. PSATO [22] 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 [10] which is the parallel implementation of SATZ [12]. Unlike GrADSAT, 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 [19] implements a different algorithm for clause sharing. In addition, PaSAT uses a *global lemma(clause) store* whereas GrADSAT shares clauses globally as soon as they are generated.

A different approach is presented by NAGSAT [6]. 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 [23]. 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 GrADSAT, this approach relies on specific hardware.

6 Conclusion:

The paper presents GrADSAT a satisfiability solver which runs on a set of widely distributed commodity computational resources. The distributed solver dynamically acquires and releases computational nodes. The solver is also 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 GrADSAT solved harder instances. Some of these instances were not solved before.

Finally, harder SAT instances can be solved by running more efficient parallel SAT solvers on more computational resources.

References

[1] 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.

[2] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Applications and Supercomputing*, 15(4):327–344, Winter 2001.

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

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

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

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

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

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

[19] 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.

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

[21] H. Zhang. SATO: An Efficient Propositional Prover. In *Proceddings of International Conference on Automated Deduction (CADE-97)*, 1997.

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

[23] 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.