NAGSAT: A Randomized, Complete, Parallel Solver for 3-SAT

Sean L. Forman and Alberto M. Segre

ABSTRACT. This extended abstract describes a 3-SAT solver, *nagsat*, that utilizes *nagging*, a new scalable, fault-tolerant, distributed search paradigm. Nagging exploits the heavy-tailed distribution of 3-SAT solution time by playing multiple reformulations of the problem or portions of the problem against each other. Nagging is a general, complete search architecture that has been applied to a variety of algorithms from the artificial intelligence literature. In this abstract, we show preliminary results for the solution of 3-SAT instances using the DPLL Algorithm and outline some of our current efforts to improve nagsat.

1. Introduction

Sound and complete SAT-solvers are typically variations on the Davis-Putnam-Logemann-Loveland procedure (DPLL) [**DLL62**]. We say it is complete because if there is a solution, the DPLL algorithm guarantees that that solution will eventually be found or it will terminate marking the problem as unsatisfiable. Variations on the DPLL algorithm typically use a heuristic value to decide which variable to branch and may implement different techniques for backtracking and unit propagation [**ZS94, LA97b**].

Other fast solution methods for satisfiability problems such as GSAT, WSAT or simulated annealing are local search procedures that are sound, but not complete [**SLM92**]. These techniques are often able to find satisfying assignments very quickly, but do not terminate conclusively on unsatisfiable problems.

Recently, "boosted" combinatorial search techniques that utilize randomized heuristic branch selection [**GSK98**] have demonstrated a way to exploit the heavy-tailed distribution of 3-SAT solution times. This technique restarts a complete solver following some number of unsuccessful backtracks. During the next run, it re-orders the search space in some way and repeats until a satisfying assignment is found or a higher backtrack threshold has been met. If the problem is unsatisfiable, eventually the number of backtracks is allowed to grow large enough that the search will terminate without finding a solution.

At SAT 2001, there were two parallel implementations of DPLL, PaSAT [**SBK01**] and //Satz [**JLU01**]. Both of these techniques can be categorized as partitioning algorithms. As a gross simplification, the main problem is broken into smaller problems and the load is balanced between the processors using some technique that passes subproblems from busy to idle processors.

Nagsat, while a parallel technique, is a departure from partitioning techniques in that it utilizes the boosting effect in a parallel manner. Using a parallel programming technique called *nagging*, nagsat supplements a basic complete search performed by a master processor at the root of the hierarchy with child processors each searching a superspace of the master's current state and each racing with the master processor to find their own solution (or lack of solution) in their own particular manner thus exploiting the heavy-tailed distribution of 3-SAT solution times without the need for restarts by the master processor.

2. Nagging and DPLL

Nagging is a general-purpose, asynchronous, parallel search technique where a single processor (or master), performing some standard search procedure, is advised by one or more nagging processors (or naggers), each performing identical search procedures on perturbed search spaces [**SFRW02**]. Nagging was originally developed in the context of DALI, a first-order logic theorem proving system [**SS94b**], and has been extended to more general search questions. It differs from most parallel search techniques in that it is not a partitioning method.

Nagging involves a single master processor and a set of subservient naggers. Nagging begins with a single master processor that alerts the available naggers to its work on a new problem and reads in the given problem. The master's responsibilities are to perform a complete DPLL search for a given problem, provide the current state of its search to naggers who ask for it, and then incorporate into its search any valuable information returned by the naggers. Currently, the master simply branches on the first free variable it happens upon.[1] This is a standard DPLL implementation with no branching heuristics [**DLL62**].

[1]This naive branching technique is clearly less than ideal, and we are currently working on implementing more state of the art branching heuristics in a random

At initialization, each nagger is provided with the original Boolean formula that specifies the 3-SAT problem the master is solving. A nagging episode begins when the nagger requests a nagpoint from the master, who briefly interrupts its own search to select a random nagpoint. Each nagpoint corresponds to one of the partial Boolean formulae currently fixed by the master's current search state. Upon receiving the nagpoint, the master resumes its own search, while the nagger first applies a problem transformation function to the search space past the nagpoint (i.e. reordering the list of as-yet-unbound variables, randomly inverting their logical sense, thereby switching the order in which V and $\neg V$ are explored, or a number of other possible techniques), and then begins its own search which is performed in the same manner as the master's search (see Figure 1). The master and the nagger now race to exhaust their respective search spaces; while the two search spaces are semantically equivalent, they will be searched differently, possibly leading to dramatically different solution times. There are only three possible outcomes to consider for a nagging event:

- **Abort:** If the master backtracks over the nagpoint before the nagger completes its own search, the master signals the nagger to abort the nagging episode, which causes the nagger to once again become idle and initiate a new nagging episode.
- **Prune:** If the nagger completes its search and finds no solution in this branch of the tree. Then the nagger interrupts the master and forces it to backtrack to the nagpoint, resulting in a reduction in the master's search space.
- **Solve:** Finally, should the nagger find a true assignment, it will terminate its own current search and report the solution to the master, who then terminates displaying the true assignment.

Of course, for nagging to have the greatest possible positive effect on the master's search efficiency, we would prefer the second and third cases occur with high probability, and we hope the first case occurs only rarely. Should the second and third cases never occur, we can expect no improvement in the master's expected time to solution: indeed, the master's search will surely be less efficient due to the small *but measurable* additional overhead of servicing its naggers. Additionally, naggers can become parent processors to other naggers allowing the parallel hierarchy to grow in a tree-like manner.

manner. Note also that this technique can be utilized on a single stand-alone machine that has spawned multiple nagging processes.

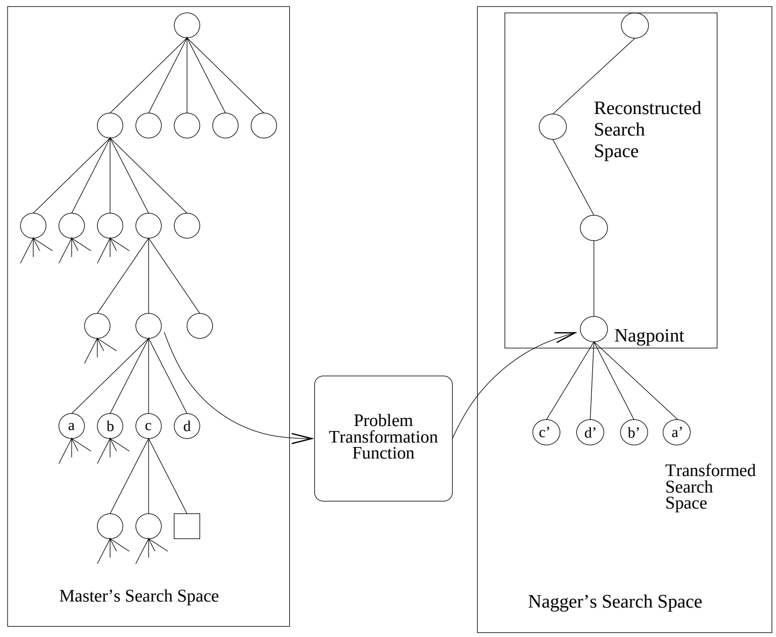

FIGURE 1. Nagging episode. The square node indicates the current position of the master process, which is executing a search in some order. A nagpoint is selected along the master's search path, and is described to the nagging process by communicating the series of choices from the root of the search tree to the nagpoint. The nagger reconstructs the master's search space up to the nagpoint and then commences exploring its own, transformed, version of the space rooted at the nagpoint.

The main advantages of nagging as a parallel technique are clear even within this simplified context. First, nagging does not require explicit load balancing, since idle machines always initiate nagging episodes to keep themselves busy. As long as the master processor is still searching, new nagpoints can be provided at will. Second, nagging is intrinsically fault tolerant (modulo the master failing): since the master's search is unaffected should a nagger fail or become inaccessible, losing a nagger will never compromise the correctness or quality of the solution produced by the master, nor will the master ever stop and wait for the missing nagger. Finally, communication is brief, since even deep search states can be described concisely as a sequence of choices, while the results reported by a nagger often reduce to a single bit (prune/abort) or just a few bits (a solution).

3. NICE: A Network Infrastructure for Combinatorial Exploration

To support our work on nagging, we have developed *NICE*, the Network Infrastructure for Combinatorial Exploration. NICE is specifically designed to support both nagging and partitioning for search algorithms; it allows application programmers to parallelize their search procedures by simply making appropriate function calls within their code. The NICE distribution includes *niced*, a Unix resource management daemon that runs on every machine, *niceq*, a daemon status query program, and *niceapi*, the applications programmer's interface library. Note that the library does not actually contain code for the search algorithms, but rather only the handful of functions that are needed to parallelize (via either nagging or partitioning) appropriately designed serial search algorithms. The code is written in ANSI C with BSD sockets over TCP/IP, and is known to run on multiple variants of the Unix operating system, including Linux, Sun OS, and HPUX.

The NICE daemon, or *niced*, fulfills three primary functions:

- The daemon maintains contact with the NICE hierarchy and re-organizes the hierarchy below the master to handle machines that may have been taken off-line and percolate faster machines to the top.
- The daemon provides the interface through which a qualifying application may request additional processors and is also responsible for security.
- The daemon manages the local host's resources according to prespecified host-specific constraints: for example, some hosts may be available only during specified times, such as during nighttime hours. This architecture is designed to run on large, heterogeneous networks of desktop PC's (for example, a series of university computer labs).

Both the NICE daemon and NICE API represent fairly mature software efforts. Versions of the NICE daemon have been running continuously on our systems for several years, with no noticeable impact on performance. The code is very robust, with NICE daemons running reliably and unobtrusively for periods of many months between system restarts.

4. Analysis and Experimental Results

In Segre (2002) [**SFRW02**], a formal analysis of general nagging methods versus partitioning and serial techniques is given. This analysis is performed on a general minimizing search. Our formal analysis relies on techniques developed for reliability data analysis; specifically, understanding how a problem transformation function used defines a probability distribution of solution times for a specific problem [**ME98**]. We used a variety of probability distributions including a heavy-tailed distribution. Our analysis showed that when performing minimizing search using nagging, the expected speedup and the probability of super-linear speedup scaled much better with nagging than with a partitioning technique.

In the case of satisfiable SAT instances minimizing search is not an appropriate model, as SAT algorithms will terminate upon locating the first solution. However, the analysis is appropriate for unsatisfiable SAT instances as those instances force the algorithm to exhaust the entire search space.

To gauge experimentally if nagging might succeed in speeding up the solution of 3-SAT instances using DPLL, we randomly generated 100 "difficult" 3-SAT problems of varying sizes [**MSL92**], 49 satisfiable and 51 unsatisfiable. For this demonstration, time to solution is measured and recorded twice for each 3-SAT instance: once using the default search ordering in serial ($t - serial$) and then a second time with naggers utilizing the random permutation transformation on the problems passed to them ($t - parallel$). The results are shown in Figure 2; each datapoint in the graph represents the CPU time required to solve in serial (x-axis) plotted against the CPU time required to solve in parallel (y-axis). These results are summarized in the table below. We have provided both the arithmetic and geometric mean (which is likely better suited to averaging ratios), along with the maximum and minimum speedups observed. Recall that speedup is solution time in serial divided by solution time in parallel. $speedup = \frac{t-serial}{t-parallel}$.

		2-processors				64-processors			
	N	min	μ	$\mu*$	max	min	μ	$\mu*$	max
satisfiable	49	0.45	12.93	2.50	221.0	0.31	42.80	12.80	653.3
unsatisfiable	51	0.98	2.36	1.62	14.2	0.99	7.03	4.68	60.9

5. Current Work and Conclusion

While the speedup is sub-linear in the case with 64 processors, we do see substantial improvement over the 2-processor case, and this would seem to imply that additional processors may continue to payoff.

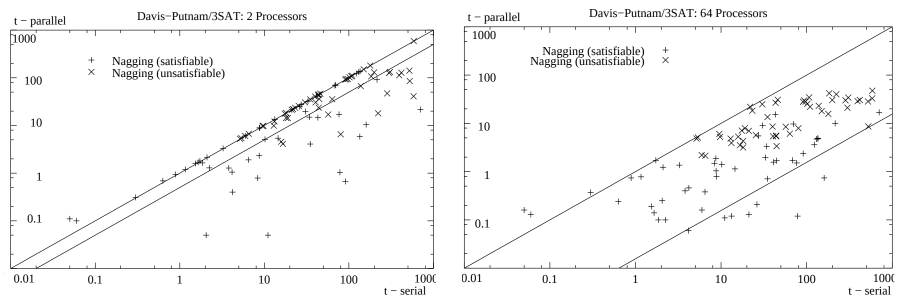

FIGURE 2. Two-processor (left) and 64-processor (right) DPLL performance in log-log space. The x-axis is the time to solve (seconds) in serial, and the y-axis is the time to solve (seconds) in parallel. Datapoints falling below the upper diagonal are faster in parallel, while datapoints below the lower diagonal line are super-linearly faster in parallel.

As mentioned in previous sections, the current DPLL solver utilized by nagging is extremely naive and was implemented essentially as a "proof of concept" for our general paper on nagging. The random ordering of variables to branch on is clearly inferior to the current state of the art branching heuristics. We will be working over the next few months on implementing various randomized branching rules within nagsat, as well as bringing other aspects of nagsat up to the current state of the art.

Even with a naive branching heuristic, nagsat represents a novel combination of parallel search and randomized boosting. It exploits the heavy-tailed distribution of 3-SAT solution times, but does so while searching the space in a complete manner on its first run and without wasting any of the master processor's search in the event of an unsatisfiable problem. It also provides an inherently fault tolerant parallel platform for the solution of satisfiability problems.

References

[DLL62] Martin Davis, George Logemann, and Donald Loveland, *A machine program for theorem-proving*, Communications of the ACM **5** (1962), no. 7, 394–397.

[GSK98] Carla P. Gomes, Bart Selman, and Henry Kautz, *Boosting combinatorial search through randomization.*, Proc. of AAAI-98 (Madison, WI), AAAI, July 1998.

[GW99] Ian P. Gent and Toby Walsh, *The search for satisfaction*, 1999.

[HV95] John N. Hooker and V. Vinay, *Branching rules for satisfiability*, Journal of Automated Reasoning **15** (1995), no. 3, 359–383.

[JLU01] Bernard Jurkowiak, Chu Min Li, and Gil Utard, *Parallelizing satz using dynamic workload balancing*, Electronic Notes in Discrete Mathematics (Henry Kautz and Bart Selman, eds.), vol. 9, Elsevier Science Publishers, 2001.

[LA97a] Chu Min Li and Anbulagan, *Heuristics based on unit propagation for satisfiability problems*, IJCAI (1), 1997, pp. 366–371.

[LA97b] Chu Min Li and Anbulagan, *Look-ahead versus look-back for satisfiability problems*, Principles and Practice of Constraint Programming, 1997, pp. 341–355.

[ME98] W. Meeker and L. Escobar, *Statistical methods for reliability data*, John Wiley & Sons, New York, NY, 1998.

[MSL92] David G. Mitchell, Bart Selman, and Hector J. Levesque, *Hard and easy distributions for SAT problems*, Proceedings of the Tenth National Conference on Artificial Intelligence (Menlo Park, California) (Paul Rosenbloom and Peter Szolovits, eds.), AAAI Press, 1992, pp. 459–465.

[SBK01] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin, *PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications*, Electronic Notes in Discrete Mathematics (Henry Kautz and Bart Selman, eds.), vol. 9, Elsevier Science Publishers, 2001.

[SFRW02] A. Segre, S. Forman, G. Resta, and A. Wildenberg, *Nagging: A scalable, fault-tolerant, paradigm for distributed search*, re-submitted (January 2002), 2002.

[SLM92] Bart Selman, Hector J. Levesque, and Dennis Mitchell, *A new method for solving hard satisfiability problems*, Proceedings of AAAI-92 (San Jose, CA), 1992.

[SS94a] A.M. Segre and D.B. Sturgill, *Using hundreds of workstations to solve first-order logic problems*, Proceedings of the Twelfth National Conference on Artificial Intelligence (Seattle, WA), AAAI, July 1994, pp. 187–192.

[SS94b] D.B. Sturgill and A.M. Segre, *A novel asynchronous parallelization scheme for first-order logic*, Proceedings of the Twelfth Conference on Automated Deduction (Nancy, France), Lecture Notes in Computer Science, vol. 814, Springer–Verlag, June 1994, pp. 484–498.

[SS97] D.B. Sturgill and A.M. Segre, *Nagging: A distributed adversarial search-pruning technique applied to first-order logic*, Journal of Automated Reasoning **19** (1997), no. 3, 347–376.

[ZS94] H. Zhang and M.E. Stickel, *Implementing the davis-putnam algorithm by tries*, Proceedings of the Fifteenth National Conference on Artificial Intelligence, 1994.

Department of Mathematics and Computer Science, Saint Joseph's University, 5600 City Ave, Philadelphia, PA 19131

E-mail address: sforman@sju.edu

Department of Management Sciences, The University of Iowa, Iowa City, IA 52242

E-mail address: segre@cs.uiowa.edu