Project Proposal

John Dickerson (jpdicker@cs.cmu.edu)
15-780: Graduate Artificial Intelligence
March 3, 2011

Scaling Today's SAT Solver

The last two decades saw an astounding increase in both the scientific capability and real-world applicability of standard SAT solvers. Breakthroughs like conflict-directed backjumping and clause learning led to well-known, open source solvers like zChaff [3] and MiniSAT [1]. This newfound power, coupled with well-documented releases of full, open source SAT software packages, has seen general solvers enter a wide berth of domains ranging from theorem proving to computational biology. As more fields become aware of the power of modern SAT solving, the variety and sizes of problems expected to be handled by SAT solvers will increase significantly. The field needs its next breakthrough.

Parallel SAT solving is one research direction that is gaining traction. There is no standard parallel SAT solver yet; the closest is a relatively untried portfolio-based parallel solver called ManySAT [2]. ManySAT provides small-scale parallelism (on the order of a single multicore consumer chip), running multiple standard DPLL runs on different cores. These runs share clauses with each other, thus reaping the benefits of both multiple tree searches and a faster mechanism for cutting down the search space. ManySAT does not, however, deal with a number of issues that will need to be handled in a truly scalable parallel SAT solver—namely communication overhead and scaling to tens, hundreds, or thousands of cores.

In my research with Tuomas Sandholm and Erik Zawadzki, I have been building a massively parallel SAT solver that combines a single-threaded DPLL tree solver with a near-limitless number of cores running other proof techniques such as hyperresolution. In its current state, this solver typically does not justify its use of so much hardware through significantly better performance. It does, however, allow clear paths around the communication overhead and small-scale parallelism barriers faced by typical parallel SAT solvers. For this project, I would like to implement a ManySAT DPLL backend to this massively parallel system. It is my hope that a small number of intelligent parallel DPLL tree searches will be able to both reap more benefit from my massively parallel proof system while providing better guidance as to where the parallel system should be spending its time. It will be interesting to study the tradeoffs between the various levels of communication (between tree searches, between hyperresolution nodes, and between tree searches and hyperresolution nodes). I also believe there is great potential to observe interesting interactions between two or more independent but complementary proof systems working in unison.

Project Plan

I provide the following plan, with 75%, 100%, and 125% cutoffs corresponding to how much I will have done by the May poster session given an overestimation, correct estimation, or underestimation of the amount of work this project will require.

75% Plan

1. Become familiar with the ManySAT system and parallel SAT solving literature.
2. Hook the ManySAT codebase into my current parallel solver; ensure that it runs with ManySAT's DPLL solver running in single-threaded mode.

3. Get ManySAT working with multiple DPLL searches in conjunction with my parallel solver.

4. Explore the tradeoffs between allocating more ManySAT cores versus more Hyperresolution cores. How does this affect the node count of the final search tree(s) versus the speed of the actual solver? Is there some obvious balance?

100% Plan

5. Create and intensively test heuristic methods for determining the level of communication between ManySAT-to-ManySAT cores, ManySAT-to-Hyperresolution, and Hyperresolution-to-Hyperresolution cores.

6. Explore how Hyperresolution handles industrial instances versus random instances. An interesting point that shows up in SAT competitions is that cutting edge SAT techniques (like conflict-directed backjumping and clause learning) provide a *much* larger boost on structured, industrial instances than their random counterparts. Any headway made in the "what works well on random SAT?" department would be a huge boon for the field.

125% Plan

7. Explore alternate parallelization tactics. For instance, while ManySAT performs multiple DPLL searches on the same (full) search space, other parallel SAT solvers perform multiple DPLL searches, each on a (disjoint, distinct) part of the search space. Perhaps there are neat interactions between Hyperresolution being used across the entire search space while, for instance, splitting the search space at a high level and then running ManySATs in each of sub-search spaces yields better results than just running one top-level ManySAT.

8. Test my ManySAT+Hyperresolution implementation on the full battery of random and industrial instances from the most recent SAT competition (parallel track). The SAT 2011 competition will not have occurred by the time this class ends, but the 2010 test battery provides a number of hard examples that are not handled by any current day solver.

References

[1] N. Eén and N. Sörensson, *An extensible SAT-solver*, Theory and Applications of Satisfiability Testing, Springer, 2004, pp. 333–336.

[2] Y. Hamadi, S. Jabbour, and L. Sais, *ManySAT: a parallel SAT solver*, Journal on Satisfiability, Boolean Modeling and Computation **6** (2009), 245–262.

[3] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: Engineering an efficient SAT solver*, Design Automation Conference, 2001. Proceedings, IEEE, 2005, pp. 530–535.