PMiniSat - A parallelization of MiniSat 2.0

Geoffrey Chu Peter J. Stuckey
Aaron Harwood
NICTA Victoria Laboratory
Department of Computer Science and Software Engineering,
University of Melbourne
email: {gchu, pjs, aaron}@csse.unimelb.edu.au

March 30, 2008

Abstract

In this poster we briefly describe some of the features of PMiniSat, a parallel SAT solver entering SAT Race 2008. PMiniSat is a parallelization of MiniSat 2.0. It features some standard parallelization techniques like dynamic work stealing using guiding paths, sharing short learnt clauses, etc. It also features an extended learnt clause sharing heuristic, global restarts, and a series of data structure changes that increases the speed of the core propagation engine by around 80%.

1 Introduction

PMiniSat is a simple parallelization of MiniSat 2.0 [1] which uses SatELite [2] as a preprocessor. Most of the features used in PMiniSat have already been previously suggested in the literature. There is only one new parallelization idea in PMiniSat. This involves sharing learnt clauses which are "short" in the context of another thread's guiding path. PMiniSat also incorporates a series of data structure improvements that were able to increase the unit propagation speed of both the sequential version of MiniSat 2.0 and PMiniSat by around 80% on average over a set of industrial instances. These changes are described in an upcoming paper and will not be discussed here, please contact the authors for more details.

2 Dynamic work stealing

PMiniSat utilises the same work stealing scheme using guiding paths as was first discussed in [3]. In PMiniSat, work is always stolen from the longest running thread, and from as high in that thread's search tree as possible. This is done so that the work granularity is large. However, stealing as high as possible still often results in chunks of work that are extremely small, e.g. 1-10 conflicts, as the search tree in industrial instances is often highly imbalanced and there are many side branches that terminate quickly. Because of this, a thread may need to perform several steals before it finds a reasonably large chunk of work. If the steal is performed on the fly by interrupting another thread, significant CPU time would be wasted waiting for threads to respond. In PMiniSat a central queue of work is kept which is topped up by the longer running threads. The idle threads can then steal from this central queue without having to wait for other threads to respond (except when the central queue runs out). This reduces the wasted CPU time caused by work stealing to negligible levels.

3 Extended clause sharing heuristic

PMiniSat utilises two clause sharing schemes. One is a general sharing scheme much like those already in the literature, e.g. [4], where a clause is shared between all threads as long as it is shorter than a certain threshold, e.g. 5 literals or less. The other sharing scheme used in PMiniSat is complementary to the first. The idea behind this heuristic is that a learnt clause's usefulness is not the same for all threads, because each thread has a different guiding path that determines its search space. For example, suppose thread 1 and thread 2 share a large part of their guiding path (i.e. thread 1 and thread 2 are searching nearby parts of the search space), whereas thread 3 has a completely different guiding path (i.e. is searching a very distant part of the search space). Then learnt clauses derived by thread 1 and thread 2 are much more useful for each other than for thread 3. To put it more concretely, suppose three threads are assigned the following guiding paths:

thread 1 - GP1: A, B, C, D
thread 2 - GP2: $A, B, C, \overline{D}$
thread 3 - GP3: $\overline{A}$, $\overline{B}$, $\overline{C}$, $\overline{D}$

Now suppose that thread 1 derives learnt clause: $\overline{A} \vee \overline{B} \vee \overline{C} \vee E \vee F \vee G$. Nominally, this learnt clause is of length 6. However, because a few of its literals are made false by the assignments in thread 1's guiding path, it's effective length given thread 1's guiding path is only 3, i.e. it behaves as the clause $E \vee F \vee G$ within thread 1's search space. Similarly, because thread 2 shares most of its guiding path with thread 1, the clause's effective length in thread 2's search space is also only 3. Whereas this clause is useless in thread 3's search space, and would be length 6 in a random part of the search space. A clause's effective length given a thread's guiding path gives a better measure of the clause's usefulness for that thread than the clauses' raw length. In PMiniSat, we share clauses which are short within the context of another thread's guiding path, but which are not necessarily short in general. This produces a hierarchical effect where threads working on a similar part of the search tree share more clauses with each other than with threads in distant parts of the search tree.

Special attention is also given to clauses with effective length of 1 within the context of a guiding path. In the sequential case, such top level unit implications are never removed from the clause database. However, in the parallel case, they can be. For example, consider the 3 guiding paths from the above example. Suppose the same thread processed all three chunks of work in the sequence GP1, GP3, GP2. Suppose that while processing GP1, we derived the clause $\overline{A} \vee \overline{B} \vee \overline{C} \vee E$. This clause has an effective length of 1 given the guiding path and is equivalent to a top level unit implication for that part of the search space. It also has effective length 1 in the context of GP2. However, if the thread first does GP3 before GP2, it is possible that this very useful learnt clause (for GP2) is pruned away before we get to GP2. PMiniSat preserves such clauses by storing them and adding them back into a thread's database when the thread steals a chunk of work with that particular guiding path.

References

[1] N. Een, N. Srensson, MiniSat - A SAT Solver with Conflict-Clause Minimization. *Proc. Theory and Applications of Satisfiability Testing (SAT'05)* (2005)

[2] N. Een, A. Biere, Effective preprocessing in SAT through variable and clause elimination. *SAT, volume 3569 of LNCS* (2005)

[3] H. Zhang, M. Bonacina, and J. Hsiang, PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation* (1996)

[4] C. Sinz, W. Blochinger, W. Kchlin, PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. *Electronic Notes in Discrete Mathematics, vol. 9* (2001)