ParSAT: Parallel DPLL SAT Solver in Haskell

Tamer Eldeeb
Department of Computer Science
Columbia University
New York, NY 10027
te2251@columbia.edu

Abstract

The Boolean Satisfiability problem (SAT) is a classical NP-Complete problem with a great number of real world applications. Being NP-Hard, there is no known polynomial time algorithm for the problem, so we have to resort to smart heuristics in order to solve large instances in practice. The DPLL algorithm is the core of many fast SAT solvers. In this project, we leverage multi-core parallelism to implement a fast DPLL-based SAT solver. Our experiments show near liner speed up on hard SAT problem instances.

1 Problem

The input of the SAT problem is a boolean formula, and the goal is to assign each variable in the formula a value of **True** or **False** such that the entire formula evaluates to **True** (i.e. is satisfied), or report that no such assignment exists. Without loss of generality we can assume that the input formula is in conjunctive normal form (CNF), i.e. made of clauses that are ANDed together where each clause is made of terms that are ORed together. Each term is either a variable or a negation of a variable.

This is a classic NP-Complete problem, and as such, there is no known polynomial time algorithm for solving it exactly. In fact, no such algorithm exists unless P=NP. However, given its huge importance in practice, many fast solvers have been developed. the DPLL algorithm forms the basis of many such solvers.

2 DPLL

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm utilizes a backtracking search approach. At each step, it chooses an unassigned variable and assigns it one of the two possible options (True or False). It then recursively evaluates the formula under this assignment, and if it fails to be satisfied it backtracks and tries the other option. It also utilizes an optimization called **Unit Propagation**. The idea behind unit propagation is that after choosing a value for a variable, if there are unsatisfied clauses in the formula that have only one unassigned term, this term must be set to True. This process continues, potentially causing a cascade of propagation which significantly limits the search space compared to a naive approach.

We implement DPLL in Haskell in the file **sequential.hs**

3 Parallelization Approach

We observe that each branch of the DPLL algorithm (i.e. assigning a variable a True or False value) can be evaluated in parallel. This makes the structure of DPLL an ideal candidate for use with the Eval monad. At each step the algorithm selects a variable then uses the rpar and rseq primitives to evaluate the two choices in parallel. The implementation is in the file **parallel.hs**

4 Evaluation

We used datasets from https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html to evaluate our parallel implementation. The experiments were run on a local iMac machine with 6 cores core i5 processor. We show results on the unsatisfiable, challenging dubois20.cnf dataset. Other datasets yielded similar results.

Cores	1	2	4	6
Runtime (seconds)	173	88	45	32

Table 1: Performance on dubois20.cnf