Parallel DPLL SAT Solver in Haskell

Jenny Pan (JP3956) and Jack Bhalla (JSB2302)

Description of Algorithm We Plan to Parallelize

We will implement a parallel Boolean satisfiability solver in Haskell using DPLL. Given a formula in conjunctive normal form (CNF), it searches for true or false values that satisfy every clause. It returns a satisfying assignment, UNSAT if none exists, or UNKNOWN on timeout. We will vary how deeply we split the search before assigning branches to workers and measure runtime, memory use, and multicore speedup.

How We Plan to Do This

We will implement Figure 1 of Zhang and Stickel (2000, p. 279) using immutable clause lists. The paper's trie and destructive implementations are outside our scope. Heule et al.'s Cube and Conquer (§4, method A; §6.2) motivates splitting at a decision-depth cutoff and solving branches in parallel. Our workers will use ordinary DPLL.

1. Parse DIMACS CNF files into lists of signed integer literals, with assignments stored separately. For example, `[1, -2]` means x_1 OR NOT x_2. Repeatedly set any literal that is alone in a clause to true, remove satisfied clauses, and delete false literals. An empty clause means contradiction; no remaining clauses means success.
2. If propagation leaves an unresolved formula, choose the lowest-numbered remaining variable and recursively try true, then false if the first branch fails. Record assignments so the solver returns a checkable solution. Check small inputs by exhaustive enumeration and compare results with MiniSat. Profile runtime and allocation, then reduce repeated list traversals and unnecessary allocation where measurements justify it.
3. To create parallel tasks, propagate and split to a configurable decision depth. Each unresolved branch becomes a task containing its simplified formula and partial assignment. Count only branching decisions, not forced assignments; depth four creates at most sixteen tasks. Discard contradictory branches and return immediately if partitioning finds a solution.
4. Put all tasks in a shared `TQueue` from `stm`. Start a fixed pool of workers with `async`; each repeatedly claims a task and runs sequential DPLL, using `evaluate` and `force` to evaluate the result inside that worker. A worker that finds SAT publishes its assignment through a `TMVar`. The coordinator returns SAT and cancels outstanding workers, or returns UNSAT after all workers finish without a solution. A timeout covering partitioning and search returns UNKNOWN and cancels the workers.
5. Compile with `-threaded` and `-rtsopts`, then test `+RTS -N1`, `-N2`, `-N4`, and `-N8` where hardware allows. Repeat elapsed-time and peak-memory measurements, including partitioning costs. Compare several cutoffs, keeping the cutoff and branching rule fixed within each scaling comparison. Compare multicore runs with one-core runs and the optimized sequential solver. Report SAT and UNSAT separately because early success changes the work performed. Equal-depth tasks may have unequal runtimes.

```
function Satisfiable (clause set S) return Boolean
    /* unit propagation */
    repeat
        for each unit clause L in S do
            /* unit subsumption */
            delete from S every clause containing L
            /* unit resolution */
            delete ¬L from every clause of S in which it occurs
        od
        if S is empty then
            return TRUE
        else if a clause becomes null in S then
            return FALSE
        fi
    until no further changes result
    /* splitting */
    choose a literal L occurring in S
    if Satisfiable (S ∪ {L}) then
        return TRUE
    else if Satisfiable (S ∪ {¬L}) then
        return TRUE
    else
        return FALSE
    fi
end function
```

Figure 1. A simple Davis–Putnam algorithm.

Figure 1 from Zhang and Stickel (2000), p. 279. The two recursive calls define independent branches that we can assign to separate workers.

Plan for Getting Input Data for the Algorithm

Use labeled SAT and UNSAT DIMACS benchmarks from SATLIB, increasing difficulty until runs are long enough to measure. Generate additional formulas with fixed random seeds and varying variable and clause counts; use MiniSat to determine expected results when it finishes. Include empty formulas, contradictory units, duplicate literals, tautologies, and malformed input. Check every returned assignment against the original formula. Never treat a timeout as UNSAT.

If time remains after completing the solver and evaluation, we may encode a small two-client arbiter using transition unrolling from Biere et al. (1999, §4). Search for an input sequence that produces simultaneous grants within a fixed number of clock cycles, then replay any counterexample. UNSAT establishes only the absence of violations within that bound and the encoding's assumptions.

References

Zhang, H., and Stickel, M. (2000). Implementing the Davis-Putnam Method. Journal of Automated Reasoning 24, 277-296. Figure 1, p. 279.

Heule, M. J. H., Kullmann, O., Wieringa, S., and Biere, A. (2012). Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. HVC 2011, LNCS 7261, 50-65. §§4 and 6.2.

Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic Model Checking without BDDs. TACAS 1999, LNCS 1579, 193-207. §4; optional circuit extension.

SATLIB. Benchmark Problems. Source of labeled DIMACS CNF instances.