COMS 4995 Project Report: A Parallel SAT Solver

Max Levatich

2020 December

Abstract

The Boolean satisfiability problem, often referred to simply as SAT, is one of the most fundamental problems in math and computer science. SAT is proven to be NP-complete (and was, in fact, the first problem to be proven NP-complete [1]), but development of more and more efficient SAT solving algorithms nevertheless continues, due to the problem's importance in computing applications such as automated planning, automated reasoning, and model checking, in addition to its theoretical interest. As it becomes increasingly difficult to make marginal improvements to existing algorithms or extract better performance from single cores, parallel implementations of SAT algorithms that can take advantage of today's massively parallel systems become more relevant. In this report, I present one such parallel implementation in Haskell of the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [2], which forms the basis of all modern SAT solvers. I give experimental results that demonstrate my algorithm's ability to take advantage of multiple cores and achieve close to ideal speedup.

1 The SAT problem

1.1 Formulation

The SAT problem in its most general formulation asks whether the variables of a given Boolean formula can be assigned to `true` or `false` in such a way that the entire formula evaluates to `true`. If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the formula is `false` for all possible variable assignments and is called unsatisfiable. For example, the formula $x \wedge \neg y$ is satisfiable if $x =$ `true` and $y =$ `false`, while the formula $x \wedge \neg x$ is unsatisfiable.

For simplicity, we assume that the Boolean formulas are given in *conjunctive normal form* (CNF), that is, as a conjunction of *clauses*, where each clause is a disjunction of *literals*, and each literal is either a boolean variable or it's negation. For example, $(x \vee y) \wedge (\neg x \vee \neg y)$ is a CNF formula, while $x \vee (y \wedge z)$ is not.

This assumption is safe to make, because every Boolean formula in propositional logic can be converted to CNF in linear time on the size of the formula [3].

1.2 The DPLL algorithm

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm has formed the backbone of every competitive SAT solver for over 50 years. Modern optimizations have not changed the core features of DPLL: *guessing backtracking*, and *unit propagation*.

The algorithm maintains a set of variable assignments which is initially empty.

The algorithm runs by *guessing* a variable value (assigning `true` or `false` to it), applying unit propagation with the addditional variable assignment, and then recursively checking if the formula is satisfiable under the guess and whatever other assignments were implied by unit propagation.

Unit propagation proceeds as follows: If, due to the current variable assignments, all literals but one in a clause are `false`, the unassigned literal must be assigned such that it is true, and this assignment can be addded to the working set. Propagation can be repeated with the new assignment, often leading to deterministic cascades of assignments, thus avoiding a large part of the naive search space. Propagation may alternatively discover that every literal in a clause is `false`, making the problem unsatisfiable. This is called a conflict.

If propagation reports a conflict, the algorithm *backtracks*, undoing assignments until it reaches the most recent guess. It then flips the value of the guess, and tries the same unit propagation and recursion. If both the original guess and its negation result in conflicts, the issue is with an earlier assignment, and the algorithm *backtracks* further. If there are ever no earlier guesses to backtrack to, the whole problem is unsatisfiable.

2 Serial implementation

My implementation of DPLL in Haskell uses a central data structure called the `Solution`, which maintains the set of current variable assignments:

```
data Choice = Guess { literal :: Int, visited :: Bool }
            | Implication { literal :: Int }
            deriving (Eq, Show)

type Trail = [Choice]
type Model = S.Set Int
```

```
type Depth = Int

type Solution = (Trail, Model, Depth)
```

A `Choice` is either a guess or an implication, i.e. something which was discovered during unit propagation; it is *implied* by the most recent guess. Choices store *literals*, which are represented by integers (a positive integer is a `true` assignment to that variable, while a negative integer is a `false` assignment). The `Trail` is a stack consisting of all of the choices made thus far. The `Model` stores the same literals as the `Trail` as a set for efficient membership checking. The `depth` stores how many guesses have been made.

The `Solution` is passed around in a `State` monad, and is updated exclusively through `push` and `pop` operations.

The core of the algorithm is captured in three functions: `dpll`, `propagate`, and `backtrack`.

```
type Clause = [Int]

dpll :: [Clause] -> State Solution Bool
dpll cs = do
    n <- next cs
    case n of
        Just i -> do
            _ <- push $ Guess i False
            propagate cs
        Nothing -> return True
```

`dpll` accepts a list of clauses, where a `Clause` is just a list of integers representing literals. The list of clauses is parsed from a DIMACS-cnf file passed to the command line.

The role of `dpll` is to guess some new unassigned variable and initiate propagation, or return `true` if there are no more variables to assign, indicating the problem is satisfiable.

```
propagate :: [Clause] -> State Solution Bool
propagate cs = do
    conflict <- unitPropagate cs
    if conflict
        then do
            continue <- backtrack
            if continue then propagate cs else return False
        else dpll cs
```

`propagate` functions mainly as a wrapper around unit propagation, that initiates backtracking if unit propagation caused a conflict, or calls `dpll` to make a new guess if there are no conflicts.

The implementation details of unit propagation are not particularly interesting, so I elide them here.

```
backtrack :: State Solution Bool
backtrack = do
    choice <- pop
    case choice of
        Just (Guess i False) -> push $ Guess (negate i) True
        Just _               -> backtrack
        Nothing              -> return False
```

`backtrack` pops assignments off the trail until it reaches a guess with the boolean value `false`, indicating that the guess has not yet been negated. It pushes the negation and returns `true` to restart propagation. It ignores already-negated guesses and any implications from unit propagation. If the trail is empty, the entire problem is unsatisfiable.

3 Parallelization

3.1 Strategies and pitfalls

There are a few pieces of the DPLL algorithm I considered parallelizing. One is unit propagation, since SAT solvers spend most of their runtime on unit propagation [4], and for a sufficiently large number of clauses, a single unit propagation is a large enough chunk of work to be worth splitting.

Unfortunately, parallelizing unit propagation turns out to require too much synchronization between threads; if clauses are partitioned between threads, and examining a clause yields a new assignment, all other threads will need to re-do their work taking the new assignment into account, since clauses might yield a new implication or conflict.

The more promising avenue is to parallelize the solving of subproblems. Every time DPLL needs to guess an assignment for `x`, it creates the subproblem of determining the satisfiability of the clauses under the new `x ++ trail`. If this subproblem returns `unsat`, DPLL tries again after flipping the variable assignment, ¬`x`.

We can instead create two subproblems with each guess: one for both assignments `x` and ¬`x`. We then execute both subproblems in parallel, backtracking only if both return `unsat`. This approach, while realizable without any communication between threads, has the disadvantage of being impossible to statically partition, because we ddon't know ahead of time which subproblems will be examined, or how long a given subproblem will take. So new subproblems must be assigned to threads dynamically.

3.2 Parallel implementation

An advantage of implementing the parallel algorithm in Haskell is that Haskell's runtime has dynamic partitioning built-in, using the `Eval` monad and `rpar` function. Provided the program is compiled with GHC's `--threaded` option, `rpar` signals to Haskell that an expression can be evaluated in parallel, using however many cores are available. Execution of the program will continue unless `rseq` is used to wait on the value of the expression.

I parallelize the solving of subproblems as follows: whenever a guess of `x` is made, first call `rpar` on a guess of ¬`x`, and then call `rseq` on the original guess. The program waits on the result of the first guess, and if it returns `sat`, ignores the negated guess (which Haskell will garbage collect). If the first guess returns `unsat`, the program waits on the result of the negated guess, which was (at least partially) evaluated in parallel with the first guess. Since the process is recursive, the number of potential sparks is $O(2^n)$, where n is the number of boolean variables.

The modifications to the code are simple enough to print in full here:

```
dpll :: [Clause] -> State Solution Bool
dpll cs = do
    n <- next cs
    case n of
        Just i -> do
            m <- model
            d <- depth
            runEval $ bothBranches m d cs i
        Nothing -> return True

bothBranches :: Model -> Int -> [Clause] -> Int -> Eval (State Solution Bool)
bothBranches m d cs i = do
    let forwSt = ([Choice i True], S.insert i m, d + 1)
        forkSt = ([Choice (negate i) True], S.insert (negate i) m, d + 1)
    fork <- rpar $ force $ runState (propagate cs) forkSt
    forw <- rseq $ runState (propagate cs) forwSt
```

```
case forw of
    (True, sol) -> return $ state (\(_, _, _) -> (True, sol))
    (False, _) -> do
        _ <- rseq fork
        case fork of
            (True, sol) -> return $ state (\(_, _, _) -> (True, sol))
            (False, _) -> return $ return False
```

4 Evaluation

4.1 Benchmark set

I relied on two benchmark sets of files encoded in the DIMACS-cnf format used in SAT competitions [5]. One set of "easy" benchmarks consists of randomly-generated satisfiable and unsatisfiable problems ranging from 25 to 2200 clauses and 10 to 180 boolean variables. These benchmarks are technically uninteresting, as most of them can be solved with only a few unit propagations and minimal backtracking, and I used this set primarily for debugging.

To evaluate the serial and parallel implementations, I used a second set of "hard" benchmarks consisting of a selection of problems from the SATLIB benchmark set (DUBOIS, PHOLE, and AIM) specifically designed to challenge SAT solvers [6].

All of the hard benchmarks are unsatisfiable; since DPLL does not use heuristics to choose literals or aggressively prune the search space, the metric of interest is how fast the parallel algorithm can enumerate a large search space, and using unsatisfiable benchmarks ensures the results do not depend on "lucky" guesses from the algorithm.

4.2 Results

All of the results reported here were achieved on a 2018 Macbook Pro with a 2.9 GHz Intel Core i9 processor and 6 logical cores. As such, my tests use at most 6 cores.

I tested a representative benchmark, `dubois20.cnf`, using different numbers of cores. The parallel implementation with 1 core performs identically to the serial version of the program, confirming that, in the absence of parallelism, their semantics are identical or nearly identical.

Performance with multiple cores achieved nearly ideal speedup, with speedup falling off slightly as the number of cores increases to the maximum (it's possible that the machine

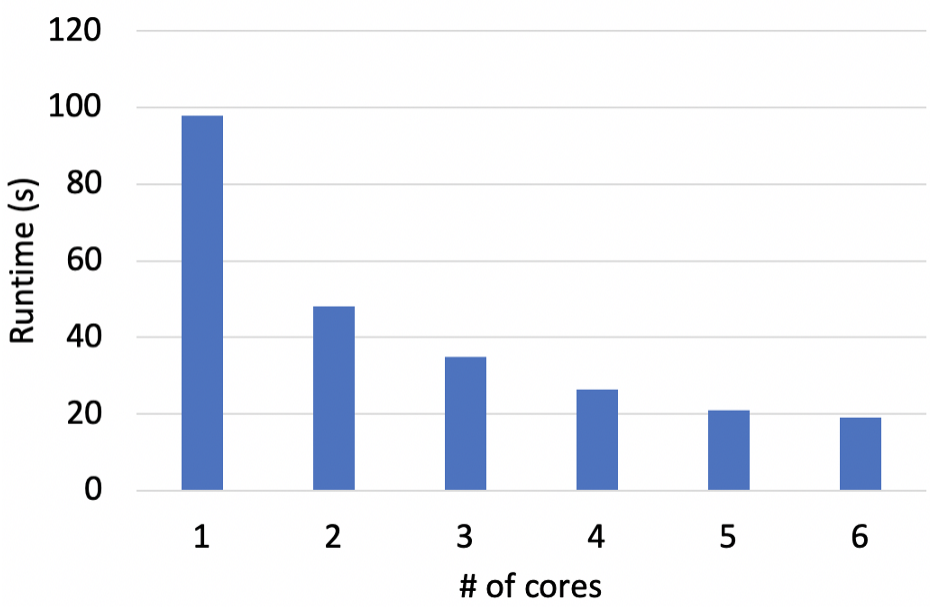

Figure 1: Performance on `dubois20.cnf` with n cores

Cores	2	3	4	5	6
Speedup	202%	277%	373%	462%	510%

Figure 2: Average speedup with n cores on `dubois20.cnf` relative to serial DPLL

needed some processing power for other applications, and that the actual average speedup for 5 or 6 cores could be higher than recorded).

Experiments showed a similar speedup on all of the hard benchmarks, for which I compared runtimes between 1 core and 6 cores. Anticipating a high number of sparks, I addditionally tracked how the runtime system was handling them.

We observe a large number of garbage collected sparks and a small number of converted sparks, which is natural given that any time an initial guess is correct, every single spark created for the negated guess is discarded. The number of converted sparks also gives us an idea of the variability in how much backtracking is required relative to the size of the SAT instance (measured as the number of boolean variables).

As the number of sparks increases exponentially as the size of the problem increases, for harder problems a limit on the trail depth at which sparks are created may be neccessary. This is an area for future exploration - although my implementation tracks guess depth, the value goes unused.

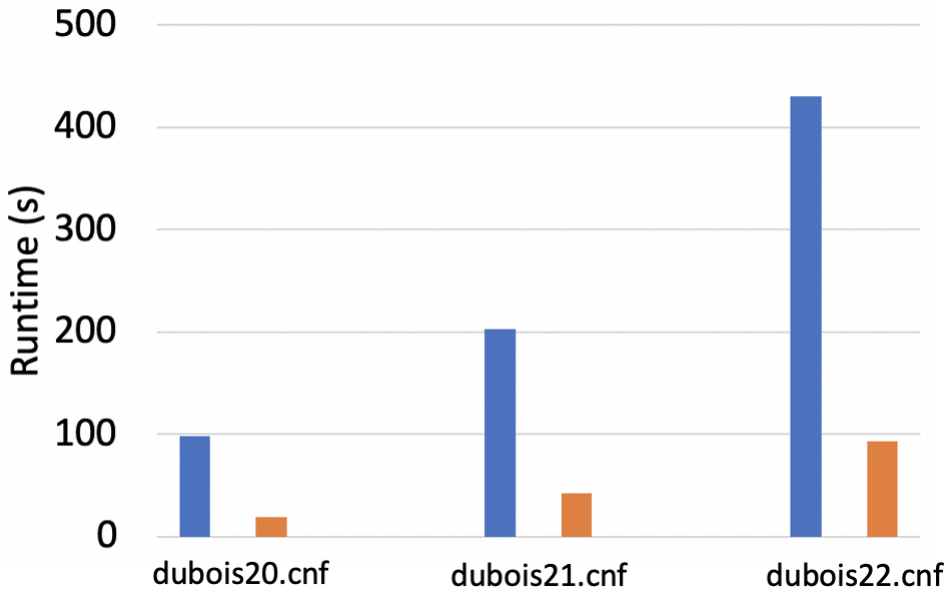

Figure 3: Serial vs 6-core performance on dubois benchmarks

Benchmark	`dubois20.cnf`	`dubois21.cnf`	`dubois22.cnf`
Size	60	63	66
Sparks	2099101	4200643	8393092
GC'd	2053111	4112034	8236800
Converted	190	518	373
Work Balance	84.96%	85.82%	86.15%

Figure 4: Runtime statistics on select benchmarks using 6 cores

Overall, given the simplicity of the change to the algorithm, the speedup exceeded my expectations. It is clear that dynamically partitioning subproblems is an incredibly effective means of parallelizing DPLL which scales to higher numbers of cores.

5 Conclusion

5.1 Summary

I implemented serial and parallel versions of the DPLL algorithm for solving the Boolean satisfiability problem in Haskell. The parallel algorithm used Haskell's Eval monad to

parallelize the serial implementation with minimally invasive changes, with creating, apportioning, and garbage collecting threads handled by the Haskell runtime.

The parallel algorithm achieved close to ideal speedup on competition benchmarks, showing the effectiveness and scalability of the parallelization.

5.2 Future Work

Modern SAT solvers use conflict-driven clause learning (CDCL), an implementation of DPLL that additionally analyzes each conflict in order to learn new clauses at runtime and perform non-chronological "back-jumping" [7]. These solvers are more powerful than standard DPLL, and a further direction for this project is to explore how CDCL can be parallelized. The addition of clause learning would require frequent synchronization between threads and introduce significant overhead to a parallel algorithm. Developing a parallel CDCL algorithm that surmounts these hurdles and incorporates other modern trappings such as portfolio-based solving would bring my work here more in line with the state-of-the-art in efficient SAT solving.

6 References

[1] https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[2] https://en.wikipedia.org/wiki/DPLL_algorithm

[3] https://en.wikipedia.org/wiki/Tseytin_transformation

[4] http://algos.inesc-id.pt/~pff/projects/parsat/publications/Costa-TR2013Feb.pdf

[5] https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps

[6] https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[7] https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf

7 Program listing

The serial and parallel implementations share most of their code, but I have included both here for completeness.

7.1 sat-serial.hs

```
import System.Environment

import Control.Monad.State

import qualified Data.List as L
import qualified Data.Set as S

------------------------------
-- Monadic assignment trail --

data Choice = Guess { literal :: Int, visited :: Bool }
            | Implication { literal :: Int }
            deriving (Eq, Show)

type Trail = [Choice]

type Model = S.Set Int
type Depth = Int
type Solution = (Trail, Model, Depth)

pop :: State Solution (Maybe Choice)
pop = state pop'
    where pop' (x@(Guess i _):xs, m, d) = (Just x, (xs, S.delete i m, d - 1))
          pop' (x@(Implication i):xs, m, d) = (Just x, (xs, S.delete i m, d))
          pop' s@([], _, _) = (Nothing, s)

push :: Choice -> State Solution Bool
push x@(Guess i _) = state (\(xs, m, d) -> (True, (x:xs, S.insert i m, d + 1)))
push x@(Implication i) = state (\(xs, m, d) -> (True, (x:xs, S.insert i m, d)))

model :: State Solution Model
model = (\(_, m, _) -> m) <$> get

printSolution :: Solution -> IO ()
printSolution (_, m, _) = putStrLn $ "Sat\n" ++ (unwords $ map show $ S.elems m)

----------------------
-- Unit propagation --
```

```
type Clause = [Int]
data Status = Safe | Conflict | Implies [Int] deriving (Eq, Show)

litStatus :: S.Set Int -> Int -> Status
litStatus m i
    | S.member i m          = Safe
    | S.member (negate i) m = Conflict
    | otherwise             = Implies [i]

clauseStatus :: S.Set Int -> Clause -> Status
clauseStatus _ []     = Conflict
clauseStatus m (i:is) = case litStatus m i of
   Safe     -> Safe
   Conflict -> continue
   imp      -> if continue == Conflict then imp else Safe
   where continue = clauseStatus m is

status :: S.Set Int -> [Clause] -> Status
status _ []     = Safe
status m (c:cs) = case clauseStatus m c of
    Safe        -> continue
    Conflict    -> Conflict
    Implies [i] -> case continue of
        Implies is -> if negate i `elem` is then Conflict else Implies (i:is)
        Safe       -> Implies [i]
        Conflict   -> Conflict
    _ -> error "bad Implies"
    where continue = status m cs

unitPropagate :: [Clause] -> State Solution Bool
unitPropagate cs = do
    m <- model
    case status m cs of
        Safe       -> return False
        Conflict   -> return True
        Implies is -> do mapM_ (push . Implication) $ L.nub is
                         unitPropagate cs

-------------------------
-- Core DPLL algorithm --
```

```
backtrack :: State Solution Bool
backtrack = do
    choice <- pop
    case choice of
        Just (Guess i False) -> push $ Guess (negate i) True
        Just _               -> backtrack
        Nothing              -> return False

propagate :: [Clause] -> State Solution Bool
propagate cs = do
    conflict <- unitPropagate cs
    if conflict
        then do
            continue <- backtrack
            if continue then propagate cs else return False
        else dpll cs

next :: [Clause] -> State Solution (Maybe Int)
next cs = do
    m <- model
    let assigned i = S.member i m || S.member (negate i) m
    return $ L.find (not . assigned) (concat cs)

dpll :: [Clause] -> State Solution Bool
dpll cs = do
    n <- next cs
    case n of
        Just i -> do
            _ <- push $ Guess i False
            propagate cs
        Nothing -> return True

-------------------------
-- Preprocessing steps --

unfalsifiable :: Clause -> Bool
unfalsifiable c = (length $ L.nub $ map abs c) /= (length c)

sanitize :: [Clause] -> [Clause]
sanitize cs = L.sortOn length $ filter (not . unfalsifiable) $ L.nub cs
```

```
-------------
-- Parsing --

cnfToClauses :: [String] -> [Clause]
cnfToClauses cnf = map (L.nub . init . map read . words) cnf

parseCnf :: String -> IO [Clause]
parseCnf f = cnfToClauses . tail . lines <$> readFile f

main :: IO ()
main = do
    args <- getArgs
    case args of
        [f] -> do
            cs <- parseCnf f
            case runState (dpll $ sanitize cs) ([], S.empty, 0) of
                (False, _) -> putStrLn "Unsat"
                (True, sol) -> printSolution sol
        _ -> putStrLn "Usage: ./sat-serial file.cnf"
```

7.2 sat-parallel.hs

```
import System.Environment

import Control.Monad.State
import Control.DeepSeq
import Control.Parallel.Strategies

import qualified Data.List as L
import qualified Data.Set as S

-----------------------------
-- Monadic assignment trail --

data Choice = Choice { literal :: Int, visited :: Bool }
            | Implication { literal :: Int }
            deriving (Eq, Show)

instance NFData Choice where
    rnf (Choice i v) = rnf i `seq` rnf v
```

```
    rnf (Implication i) = rnf i

type Trail = [Choice]
type Model = S.Set Int
type Solution = (Trail, Model, Int)

pop :: State Solution (Maybe Choice)
pop = state pop'
    where pop' (x@(Choice i _):xs, m, d) = (Just x, (xs, S.delete i m, d - 1))
          pop' (x@(Implication i):xs, m, d) = (Just x, (xs, S.delete i m, d))
          pop' s@([], _, _) = (Nothing, s)

push :: Choice -> State Solution Bool
push x@(Choice i _) = state (\(xs, m, d) -> (True, (x:xs, S.insert i m, d + 1)))
push x@(Implication i) = state (\(xs, m, d) -> (True, (x:xs, S.insert i m, d)))

model :: State Solution Model
model = (\(_, m, _) -> m) <$> get

depth :: State Solution Int
depth = (\(_, _, d) -> d) <$> get

printSolution :: Solution -> IO ()
printSolution (_, m, _) = putStrLn $ "Sat\n" ++ (unwords $ map show $ S.elems m)

----------------------
-- Unit propagation --

type Clause = [Int]
data Status = Safe | Conflict | Implies [Int] deriving (Eq, Show)

litStatus :: S.Set Int -> Int -> Status
litStatus m i
    | S.member i m          = Safe
    | S.member (negate i) m = Conflict
    | otherwise             = Implies [i]

clauseStatus :: S.Set Int -> Clause -> Status
clauseStatus _ []     = Conflict
clauseStatus m (i:is) = case litStatus m i of
   Safe     -> Safe
```

```
   Conflict -> continue
   imp      -> if continue == Conflict then imp else Safe
   where continue = clauseStatus m is

status :: S.Set Int -> [Clause] -> Status
status _ []     = Safe
status m (c:cs) = case clauseStatus m c of
    Safe        -> continue
    Conflict    -> Conflict
    Implies [i] -> case continue of
        Implies is -> if negate i `elem` is then Conflict else Implies (i:is)
        Safe       -> Implies [i]
        Conflict   -> Conflict
    _ -> error "bad Implies"
    where continue = status m cs

unitPropagate :: [Clause] -> State Solution Bool
unitPropagate cs = do
    m <- model
    case status m cs of
        Safe       -> return False
        Conflict   -> return True
        Implies is -> do mapM_ (push . Implication) $ L.nub is
                         unitPropagate cs

-------------------------
-- Core DPLL algorithm --

backtrack :: State Solution Bool
backtrack = do
    choice <- pop
    case choice of
        Just (Choice i False) -> push $ Choice (negate i) True
        Just _                -> backtrack
        Nothing               -> return False

propagate :: [Clause] -> State Solution Bool
propagate cs = do
    conflict <- unitPropagate cs
    if conflict
        then do
```

```
            continue <- backtrack
            if continue then propagate cs else return False
        else dpll cs

next :: [Clause] -> State Solution (Maybe Int)
next cs = do
    m <- model
    let assigned i = S.member i m || S.member (negate i) m
    return $ L.find (not . assigned) (concat cs)

bothBranches :: Model -> Int -> [Clause] -> Int -> Eval (State Solution Bool)
bothBranches m d cs i = do
    let forwSt = ([Choice i True], S.insert i m, d + 1)
        forkSt = ([Choice (negate i) True], S.insert (negate i) m, d + 1)
    fork <- rpar $ force $ runState (propagate cs) forkSt
    forw <- rseq $ runState (propagate cs) forwSt
    case forw of
        (True, sol) -> return $ state (\(_, _, _) -> (True, sol))
        (False, _) -> do
            _ <- rseq fork
            case fork of
                (True, sol) -> return $ state (\(_, _, _) -> (True, sol))
                (False, _) -> return $ return False

dpll :: [Clause] -> State Solution Bool
dpll cs = do
    n <- next cs
    case n of
        Just i -> do
            m <- model
            d <- depth
            runEval $ bothBranches m d cs i
        Nothing -> return True

-------------------------
-- Preprocessing steps --

unfalsifiable :: Clause -> Bool
unfalsifiable c = (length $ L.nub $ map abs c) /= (length c)

sanitize :: [Clause] -> [Clause]
```

```
sanitize cs = L.sortOn length $ filter (not . unfalsifiable) $ L.nub cs

-------------
-- Parsing --

cnfToClauses :: [String] -> [Clause]
cnfToClauses cnf = map (L.nub . init . map read . words) cnf

parseCnf :: String -> IO [Clause]
parseCnf f = cnfToClauses . tail . lines <$> readFile f

main :: IO ()
main = do
    args <- getArgs
    case args of
        [f] -> do
            cs <- parseCnf f
            case runState (dpll $ sanitize cs) ([], S.empty, 0) of
                (False, _) -> putStrLn "Unsat"
                (True, sol) -> printSolution sol
        _ -> putStrLn "Usage: ./sat-parallel file.cnf"
```