Parallel SAT Solver with DPLL

Wonhyuk (Harry) Choi (wc2737) and Andrew Phillipe de Soler (apd2149)

1 Introduction

This project is a parallel SAT Solver that will solve boolean satisfiability problems. The SAT Solver implements a backtracking algorithm that will keep assigning variables in the expression a boolean value until the problem is deemed satisfiable or unsatisfiable. Additionally, the SAT Solver implements the DPLL algorithm, which improves the performance of the backtracking algorithm by adding a set of heuristics, which are Literal Elimination and Unit Propagation.

As the SAT Solver assigns variables a boolean value in the expression until the problem's satisfiability is determined, the SAT Solver implements parallelism by computing the value of the boolean expression when a variable is assigned a True and False Boolean value in parallel. With four cores, we see an increase in performance of 2.33 over the sequential implementation. The Sat Solver sees the highest performance increase per thread with two cores. Performance after two cores becomes less significant.

The SAT Solver was tested against the SATLIB Benchmarks Problems from the University of British Columbia. We used at most four cores to evaluate our algorithm.

2 SAT Solvers

The purpose of an SAT Solver is to determine whether a boolean expression is satisfiable or unsatisfiable. In other words, it is satisfiable if there is an assignment of True and False values to variables in the boolean expression that will lead to the expression evaluating to True. If there is no assignment that can lead the boolean expression to evaluate to True, then the boolean expression is unsatisfiable. SAT is an NP-complete problem and was the first problem to be proven to be NP-complete.

SAT Solvers have a variety of uses. They can be used for hardware/software verification, cryptography, artificial intelligence, and more. Sat Solvers have solved scheduling problems, tested pattern generation, and aided in cryptanalysis.

3 Implementation

3.1 I/O Parsing

The SATLIB library comes in DIMACS format.

```
c Example CNF format file
c
p cnf 4 3
1 3 -4 0
4 0 2
-3
```

In order to convert the DIMACS file to our algorithm input format, we used the `parse-dimacs` library [2]. The library returns `UArrays`, which we wrote a wrapper to transform into our algebraic data type.

3.2 Sequential Algorithm

Given a CNF boolean expression in DIMACS format, the SAT Solver will print out whether the problem is satisfiable or unsatisfiable. The SAT Solver is based off of Andrew Gibiansky's blog article [1]. It implements a backtracking search algorithm to determine the satisfiability of a given boolean expression. After the SAT Solver receives the boolean expression in an input format the SAT Solver is able to process, the SAT Solver will find the first variable in the boolean expression that has not been assigned a boolean value (a free variable). Next, the SAT Solver will guess that this free variable is True. After this guess is made, every occurrence of this free variable is given the value True. Subsequently, the SAT Solver will attempt to simplify the expression and see if the expression can evaluate to True or False. If the expression cannot be simplified, the backtracking algorithm will recurse and proceed to find another free variable to assign it a boolean value of True. On the other hand, if the boolean expression can be simplified to True, then the SAT Solver will state on the command line that the boolean expression is "SATISFIABLE". On the other hand, if the boolean expression simplifies to False, the SAT Solver will take the most recent variable that was assigned True and assign the value False. If the problem simplifies to False after assigning a variable the boolean value False, then the boolean expression is "UNSATISFIABLE". Note Figure for an illustrated example.

In order to improve the performance of the recursive backtracking algorithm, the SAT solver implements the DPLL algorithm (Davis-Putnam-Logemann-Loveland). In a nutshell, the DPLL algorithm adds heuristics that improve the performance of the backtracking algorithm. It is an algorithm that requires the boolean expression to be in CNF form. The heuristics added are Literal Elimination and Unit Propagation.

Literal Elimination determines the polarity of the variables in a boolean expression. Suppose that a variable in "A" appears in only the negative polarity. In other words, in the boolean expression A is only seen as !A. The Literal

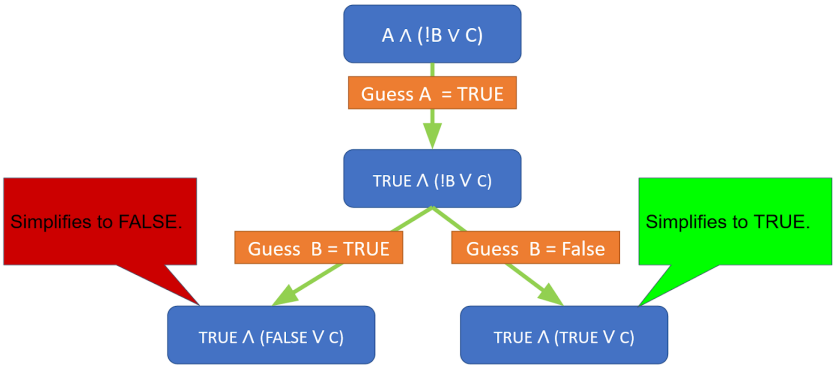

Fig. 1: Illustrated example of backtracking algorithm without heuristics to improve performance

Elimination heuristic would deduce that the value of A is False. On the other hand, if "A" appears only in positive polarity (just as "A"), literal elimination would indicate that the value is True.

In the implementation of Literal Elimination, the program places all variables in the boolean expression into a list where each unique variable will appear once in this list of variables (hereinafter a "literal list"). Afterwards, a function to determine whether a variable has a positive, negative, or mixed polarity is mapped over this literal list (hereinafter a "polarity list"). The polarity list and literal list are `zipWith`ed with a function that will produce a list of tuples that will indicate the boolean value that each variable in the literal list should be assigned. Variables that are in a Positive Polarity would be assigned True, Negative Polarity would be assigned False, and mixed would not be assigned a boolean value (an assignment list). The program would use this list of assignments to assign each occurrence of the variable in the boolean expression with the corresponding boolean value determined in the list of assignments.

Furthermore, Unit Propagation increases the backtracking performance by searching the entire Expression for clauses that contain only one variable. When such a clause is found, it will assign either a True or False value to every occurrence of the variable in the boolean expression.

The implementation of the Unit Propagation algorithm is constructed as follows. The first step is to break the boolean expression into a list of clauses. Since every clause is AND'ed together, it is a matter of traversing the AND's and creating a list of them (clause list). Subsequently, the program takes out all the clauses of the clause list that contain only one variable (a unit clause) and indicate in a tuple whether the variable is assigned a boolean value of True or False. If the `unitClause` contains a variable that is NOT'ed, then the variable should be the value False. Otherwise, the variable should be assigned the value True. This will generate a list of tuples that contains the variable that needs to

be assigned a boolean value and the value the variable should have (an assignment list). Finally the program will take this assignment list and replace every occurrence of the variable from the list that appears in the boolean expression with the corresponding boolean value in the assignment list.

Prior to the SAT Solver finding a free variable in the boolean expression to assign a Boolean value to, the variables in the expression will first be assigned boolean values with Unit Propagation. Afterwards, the variables in the expression will be assigned Boolean values through Literal Elimination. Thus prior to the SAT Solver finding a free variable in order to guess if the variable should be True or False, DPLL algorithm assigns boolean values to as many variables that fit the conditions of the heuristic.

3.3 Parallel Algorithm

We tried various methods to parallelize our sequential algorithm, including the simplification of expressions, guessing multiple free variables simultaneously, and parallelizing the false and true guesses for each variable.

Ultimately, we found that only parallelizing the boolean guesses led to appreciable performance increase. Parallelizing the expression simplification led to far too great of spark creation with the vast majority being garbage collected. Guessing multiple free variables ended up repeating a lot of work that we already computed in another guess. If we had implemented a stateful conflict-driven clause learning (CDCL) algorithm, we could have taken advantage of this parallelization, but without it, we found the algorithm to instead more overhead than its sequential counterpart.

Eventually, we focused on fine-tuning our parallelization of true/false guess of each variable. An illustrated example in Figure 2 explains the process. For each time we make a guess on a free variable, we divide the guess into two branches, a true branch and a false branch. Each branch needs to do additional work by guessing extra variables, and for each recursive call of the procedure, we parallelize the work to a thread.

We found out that we needed to tune parameters to optimize performance of our algorithm. First, we found that parallelizing only to Weak Head Normal Form using `rpar` improved performance more than evaluating completely with `rparWith rdeepseq`. We also found that we needed to limit the recursion depth to designate threads to guesses, or we would create too many sparks that would dud or get garbage collected. The optimal depth differed on each machine, but on the VM we tested on, we found a depth of 40 to show best empirical performance.

While we have only included the guess parallelization code in this report, the other parallelization attempts are available as separate branches on the github repository of the project at
`https://github.com/wonhyukchoi/parallel-sat-solver`.

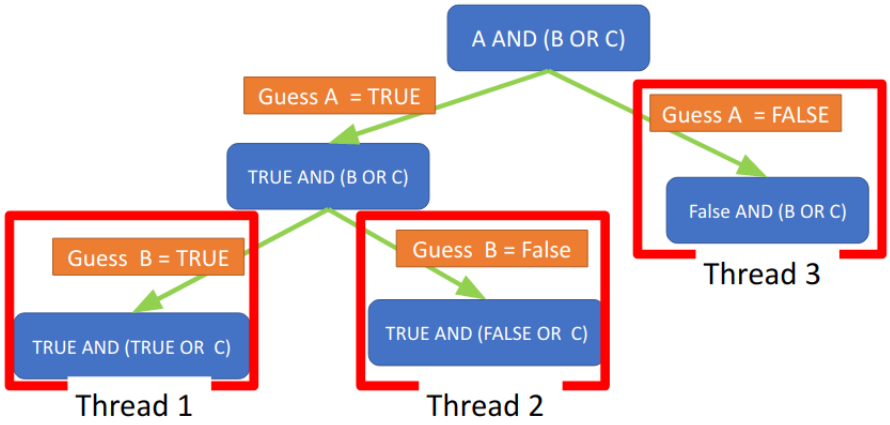

Fig. 2: Illustrated example of the variable false/true guess parallelization.

4 Evaluation

4.1 Sequential v.s. Parallel

Our evaluation used the SATLIB benchmark as produced by the University of British Columbia. In particular, we mainly used the pigeonhole problem set, as they had a reasonable amount of variables (42∼110) and clauses (133∼561), giving us a practical dataset to measure our performance.

For evaluations, we used the `n1-standard-4` virtual machine on Google Cloud Platform. The VM had an Intel Xeon CPU@2.30Ghz with 4 cores, 15Gb of RAM, and ran `Ubuntu 18.04 LTS`. We measured our runtime on $N = 1 \ldots 4$ threads, and ran 10 iterations to obtain an accurate result. Instead of calculating the mean of the 10 iterations, we took the lowest runtime of each thread test runtime, as the lowest value often gives the lower bound for the algorithm. The higher values are not caused by the variability by the algorithm, rather by other processes and scheduling on the machine.

Regardless of the problem set, we obtained very similar results. In general, we saw a marketed increase in performance when we increased the number of threads from $N = 1$ to $N = 2$, but our performance did not increase significantly afterwards, as seen on Figure 3.

In our best-case analysis, we improve sequential performance by 2.33 times on four threads over our sequential version.

A threadscope analysis of the algorithm shows that the four cores evenly divide the workload. Initially, the sequential part of the algorithm moves the file from I/O and parses it, and distributes it to each thread, but parallelization performs rather evenly afterwards. Figure 4 visualizes the results.

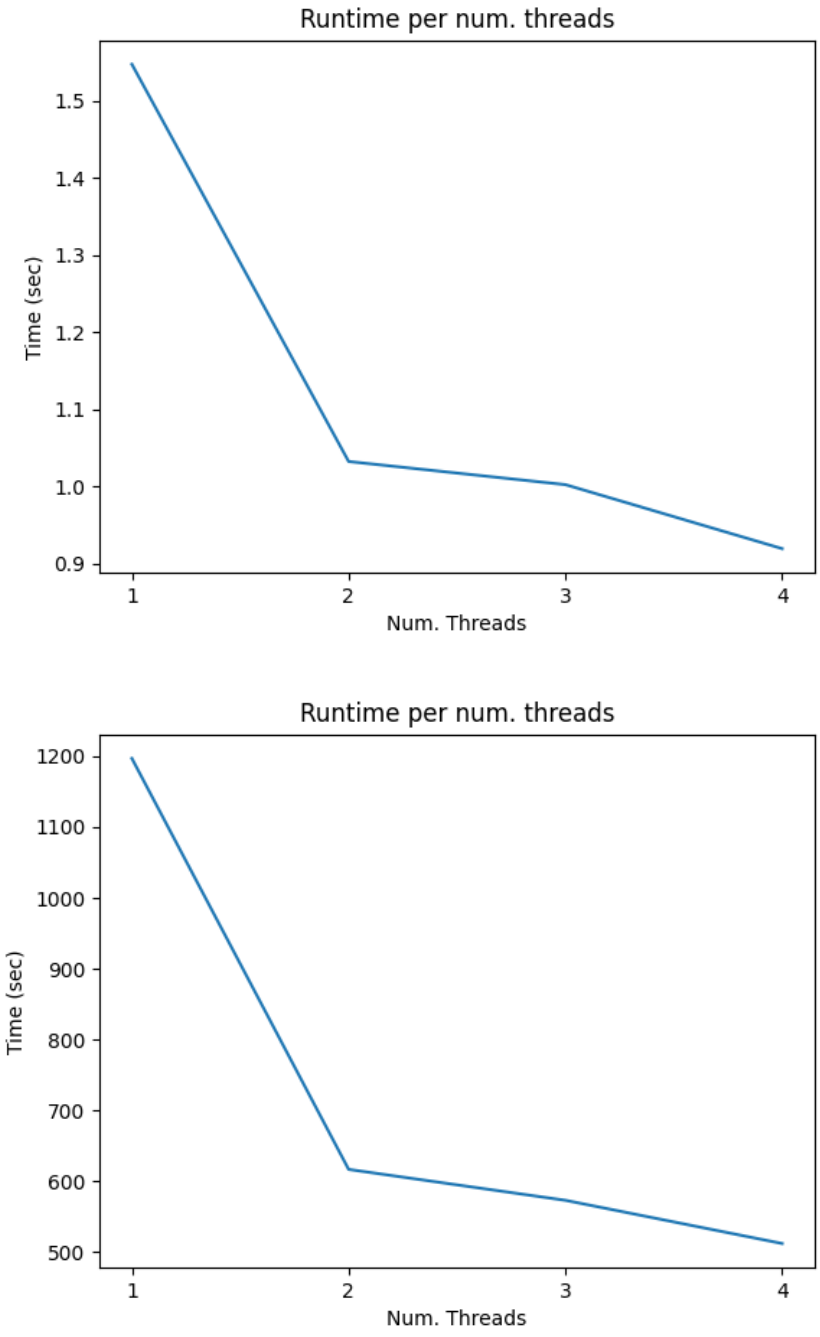

Fig. 3: Runtime evaluation for two different problems.

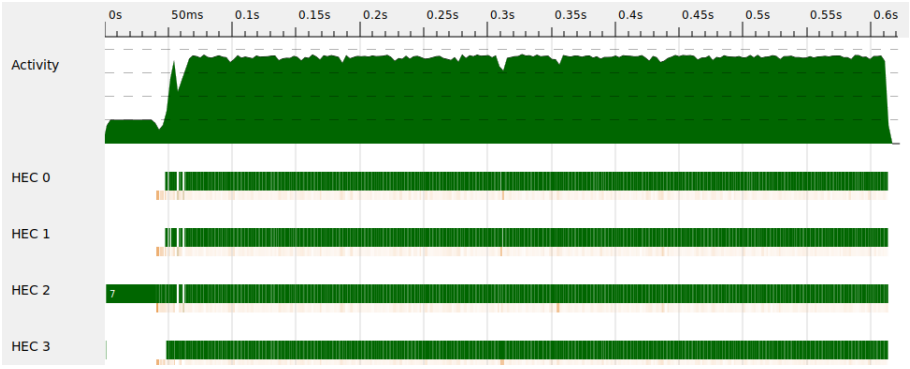

Fig. 4: Threadscope analysis of the parallel algorithm on four threads.

4.2 Comparison to SotA solvers

Unfortunately, it is difficult to measure runtime performance against State-of-the-Art(SotA) SAT solvers for three main reasons.

First, the SATLIB library, while well organized, has ceased maintenance since 2011, and modern SAT solver competitions measure SAT solver performance by requiring participants to submit their own benchmarks [3]. As we have used the well-organized SATLIB library, we cannot compare our runtimes.

Second, benchmark performance of SotA solvers only show performance for benchmarks of size incomparable to our example. While our runtime jumped from sub-second performance on a problem with 42 variables and 133 clauses to nearly ten minutes on a problem with 110 variables and 561 variables, SotA solvers are measured against giant problems with over 10^6 variables and 10^7 clauses [3].

Third, the way SotA solvers are measured are different from our time measurement evaluation. SAT solvers are typically measured by the number of test cases they can solve, and ranked descending by the number of cases terminated. Therefore, results of SotA solvers are in a format incompatible with our results.

Nonetheless, given the large size of problems SotA solvers solve under an hour, we can assume that the disparity between our program and SotA algorithms is exponential.

5 Lessons Learned

Ultimately, we found that expressing our boolean expression (no pun intended) inductively led to difficult parallelization. A boolean expression in CNF is a large AND clause of multiple clauses. A list implementation of the clauses may make it easier to parallelize, as we could spark each element of the list instead of traversing a recursive data structure. The inductive type also makes it difficult to debug, as threads are solving different levels of the problem concurrently.

Our inductive structure, and parallel backtracking algorithm also ended up being poorly parallelized because the program threw away a lot of work. The core of our implementation depended on AND and OR statements. In an AND statement, if we get that one operand is False, then we do not need the other operand. However, in our parallelization, we often spark work for the unused operand, and therefore waste computation time.

On a similar vein, limits of short-circuiting operators also limited our parallelization. In a code snippet `AND True False`, the program evaluates `True` before `False`. However, evaluation of `True` may not be needed if the execution of `False` finished first, however there does not seem to reliably force the evaluation of an AND statement to short circuit depending on the thunk that finishes first.

6 Code Listing

6.1 app/Main.hs

```
module Main where

import ParSolver
import ParseIO

import System.Exit(die)
import System.Environment(getArgs, getProgName)


main :: IO ()
main = do
      args <- getArgs
      case args of
            [file] -> do
                  cnf <- loadFile file
                  case cnf of
                        Left err   -> putStrLn err
                        Right cnf' -> case satisfiableDPLL cnf' of
                              True -> putStrLn $ file ++
                                    " is SATISFIABLE"
                              _    -> putStrLn $ file ++
                                    " is UNSATISFIABLE"
            _ -> do
                  pn <- getProgName
                  die $ "Usage: " ++ pn ++ " <filename>"
```

6.2 src/ParseIO.hs

```
module ParseIO where
import Lib(Expr(..))
```

```haskell
import Data.Array.Unboxed
import Language.CNF.Parse.ParseDIMACS

loadFile :: FilePath -> IO (Either String Expr)
loadFile fileName = do
  dimacsCNF <- parseFile fileName
  case dimacsCNF of
    Left _ -> return $ Left "Parse Error"
    Right cnf -> return $ Right $ cnfToExpr cnf

cnfToExpr :: CNF -> Expr
cnfToExpr cnf = andClauses exprList
  where
    exprList   = map uArraytoExpr $ clauses cnf

    andClauses :: [Expr] -> Expr
    andClauses []      = Const True
    andClauses [x]     = x
    andClauses [x,y,z] = And (And x y) z
    andClauses xs      = And (andClauses front) (andClauses back)
      where (front, back) = splitAt ((length xs + 1) `div` 2) xs

-- Converts UArray (DIMACS format) to our ADT
uArraytoExpr :: UArray Int Int -> Expr
uArraytoExpr = orLiterals . (map intToExpr) . elems
  where
    intToExpr :: Int -> Expr
    intToExpr n | n > 0 = Var $ show n
    intToExpr n         = Not $ Var $ show $ abs n

    orLiterals :: [Expr] -> Expr
    orLiterals = foldr1 (\x acc -> Or x acc)
```

6.3 src/Lib.hs

```haskell
module Lib where

import Control.Applicative ((<|>))
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Maybe (mapMaybe, catMaybes)

data Expr = Var String
   | And Expr Expr
   | Or Expr Expr
   | Not Expr
```

```
   | Const Bool
  deriving (Show, Eq)

-- Return the first free variable in the expression.
freeVariable :: Expr -> Maybe String
freeVariable (Const _) = Nothing
freeVariable (Var v) = Just v
freeVariable (Not e) = freeVariable e
freeVariable (Or x y) = freeVariable x <|> freeVariable y
freeVariable (And x y) = freeVariable x <|> freeVariable y

guessVariable :: String -> Bool -> Expr -> Expr
guessVariable var val expr =
   case expr of
      Var v -> if v == var
               then Const val
               else Var v
      Not expr' -> Not (guess expr')
      Or x y -> Or (guess x) (guess y)
      And x y -> And (guess x) (guess y)
      Const b -> Const b
   where guess = guessVariable var val

-- Recursively evaluate the expression until we arrive at
-- the Variable or a Boolean Value
simplify :: Expr -> Expr
simplify (Const b) = Const b
simplify (Var v) = Var v
simplify (Not expr) =
  case simplify expr of
     Const b -> Const (not b)
     expr' -> Not expr'
simplify (Or x y) =
   let es = filter (/= Const False) [simplify x, simplify y] in
        if Const True `elem` es
        then Const True
        else
          case es of
            []       -> Const False
            [e]      -> e
            [e1, e2] -> Or e1 e2
            _        -> error "Should never happen."
simplify (And x y) =
   let es = filter (/= Const True) [simplify x, simplify y] in
        if Const False `elem` es
```

```
        then Const False
        else
          case es of
            []       -> Const True
            [e]      -> e
            [e1, e2] -> And e1 e2
            _        -> error "Should never happen."

-- Unwrap the Boolean from the DataType
unConst :: Expr -> Bool
unConst (Const b) = b
unConst _ = error "Not Const"

-- Remove Negations - apply De Morgan's Law
fixNegations :: Expr -> Expr
fixNegations expr =
   case expr of
      Not (Not x) -> fixNegations x
      Not (And x y) -> Or (fixNegations $ Not x) (fixNegations $ Not y)
      Not (Or x y) -> And (fixNegations $ Not x) (fixNegations $ Not y)
      Not (Const b) -> Const (not b)
      Not x -> Not (fixNegations x)
      And x y -> And (fixNegations x) (fixNegations y)
      Or x y -> Or (fixNegations x) (fixNegations y)
      x -> x

-- Unwrap the Literals in the expression
literals :: Expr -> Set String
literals (Var v) = Set.singleton v
literals (Not e) = literals e
literals (And x y) = Set.union (literals x) (literals y)
literals (Or x y) = Set.union (literals x) (literals y)
literals _ = Set.empty

data Polarity = Positive | Negative | Mixed deriving (Show, Eq)

-- Find the polarities of the literals in the expression
literalPolarity :: Expr -> String -> Maybe Polarity
-- positive polarity
literalPolarity (Var v) v'
    | v == v' = Just Positive
    | otherwise = Nothing
-- negative polarity
literalPolarity (Not (Var v)) v'
    | v == v' = Just Negative
```

```
    | otherwise = Nothing
-- recursively find polarities in And and Or constructors
literalPolarity expr v =
   case expr of
      And x y -> combinePolarities [x, y]
      Or x y  -> combinePolarities [x, y]
      Not x   -> error $ "Not in CNF: negation of a non-literal: " ++ show x
      Const _ -> Nothing
      _       -> error "Should never happen."
   where
      combinePolarities es =
        let polarities = mapMaybe (flip literalPolarity v) es
        in case polarities of
          [] -> Nothing
          ps -> if all (== Positive) ps
                then Just Positive
                else if all (== Negative) ps
                     then Just Negative
                     else Just Mixed


literalElimination :: Expr -> Expr
literalElimination e =
    let ls = Set.toList (literals e)
        ps = map (literalPolarity e) ls

        -- Determine Polarity that needs to be assigned to the Literal
        extractPolarized :: String -> Maybe Polarity -> Maybe (String, Bool)
        extractPolarized v (Just Positive) = Just (v, True)
        extractPolarized v (Just Negative) = Just (v, False)
        extractPolarized _ _ = Nothing

        -- Gives you all the Polarity Assignments of each Literal
        assignments :: [(String, Bool)]
        assignments = catMaybes $ zipWith extractPolarized ls ps

        -- Replace the literals with a Boolean Value
        replacers :: [Expr -> Expr]
        replacers = map (uncurry guessVariable) assignments
        replaceAll :: Expr -> Expr
        replaceAll = foldl (.) id replacers
    in replaceAll e

-- Find the clauses where there is only 1 literal in the clause
unitClause :: Expr -> Maybe (String, Bool)
```

```haskell
unitClause (Var v) = Just (v, True)
unitClause (Not (Var v)) = Just (v, False)
unitClause _ = Nothing

-- Create a list of clauses by travesing the tree of And constructors
clauses :: Expr -> [Expr]
clauses (And x y) = clauses x ++ clauses y
clauses expr = [expr]

-- Extract all unit clauses
allUnitClauses :: Expr -> [(String, Bool)]
allUnitClauses = mapMaybe unitClause . clauses

-- this will replace all unit clauses with the appropiate Boolean Value
unitPropagation :: Expr -> Expr
unitPropagation expr = replaceAll expr
    where
      assignments :: [(String, Bool)]
      assignments = allUnitClauses expr
      replaceAll :: Expr -> Expr
      replaceAll = foldl (.) id (map (uncurry guessVariable) assignments)
```

6.4 src/ParSolver.hs

```haskell
module ParSolver where

import Lib
import Control.Parallel.Strategies(Strategy, using, rpar)
import Control.DeepSeq(NFData)

-- Wrapper for parallelizing the problem
satisfiable :: Expr -> Bool
satisfiable (Const b) = b
satisfiable orExpr@(Or _ _) = satisfiableDPLL orExpr
satisfiable (And x y) = and ([satisfiableDPLL x, satisfiableDPLL y]
                        `using` pairParStrat)
satisfiable _ = undefined

satisfiableDPLL :: Expr -> Bool
satisfiableDPLL = satBase pairParStrat fixedDepth

pairParStrat :: (NFData a)=> Strategy [a]
pairParStrat [a,b] = do
    a' <- rpar a
    b' <- rpar b
    return [a', b']
```

```haskell
pairParStrat _ = undefined

fixedDepth :: Int
fixedDepth = 40

-- In Parallel, evaluate TrueGuess and FalseGuess
-- recurses to a certain depth
satBase :: Strategy [Bool] -> Int -> Expr -> Bool
satBase _ 0 expr = satisfiableDPLLSeq expr
satBase strat depth expr =
    case freeVariable expr' of
        Nothing -> unConst $ simplify expr'
        Just v ->
          let trueGuess  = satBase strat depth' $
                           simplify (guessVariable v True expr')
              falseGuess = satBase strat depth' $
                           simplify (guessVariable v False expr')
          in or ([trueGuess, falseGuess] `using` strat)
    where
        depth' = depth - 1
        expr' = literalElimination $ fixNegations $
                unitPropagation expr

-- sequential satisfiable function after the depth has been reached
satisfiableDPLLSeq :: Expr -> Bool
satisfiableDPLLSeq expr =
    case freeVariable expr' of
        Nothing -> unConst $ simplify expr'
        Just v ->
          let trueGuess = simplify (guessVariable v True expr')
              falseGuess = simplify (guessVariable v False expr')
          in satisfiableDPLLSeq trueGuess ||
             satisfiableDPLLSeq falseGuess
    where
    -- Apply our backtracking search *after* literal elimination
    -- and unit propagation have been applied!
    expr' = literalElimination $ fixNegations $
            unitPropagation expr
```

References

1. Gibiansky, F. Writing a SAT Solver. (2015) https://andrew.gibiansky.com/blog/verification/writing-a-sat-solver/.
2. Bueno, D. parse-dimacs: DIMACS CNF parser library. (2012) https://hackage.haskell.org/package/parse-dimacs/.

3. Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. Proceedings of SAT COMPETITION 2018: Solver and Benchmark Descriptions. (2018) https://helda.helsinki.fi/bitstream/handle/10138/237063/sc2018_proceedings.pdf?sequence=6&isAllowed=y/.

7 Postscript

This document was formatted using the Lecture Notes in Computer Science guide by Springer. It's in LaTeX, so it must be true.