Introduction
SAT is the problem of answering the question whether or not a propositional formula is satisfiable. Despite the worst-case exponential run time of all known algorithms, satisfiability solvers are increasingly leaving their mark as a general-purpose tool in areas as diverse as software and hardware verification. More concretely, as the existing sequential algorithm used the Davis-Putnam-Logemann-Loveland (DPLL) algorithm which decides satisfiability of propositional formula in clause normal form, by using unit propagation and case distinction. This project’s goal is parallelizing the DPLL algorithm in the functional programming language Haskell. I would like to find the performance increments between sequential DPLL algorithm and DPLL algorithm after parallelization under 1-4 core VM.

DPLL
DPLL is essentially a depth-first search, which alternates between three strategies. At any stage of the search, there are partial assignments (that is, assigning values to a certain subset of variables) and a set of undecided clauses (that is, unsatisfied clauses).

1) The first strategy is pure literal elimination: if an unassigned variable \( x \) only appears in a set of unsubscribing sentences in its positive form (that is, the literal \( \neg x \) does not appear anywhere), then we can use it in our assignment Add \( x = true \) and satisfy all clauses containing the literal \( x \) (again, if \( x \) only appears in the negative form, \( \neg x \), we can add \( x = false \) to our assignment).

2) The second strategy is unit propagation: if all literals except one literal in the undecided sentence are false, the remaining literals must be true. If the remaining text is \( x \), we add \( x = true \) to our assignment; if the remaining text is \( \neg x \), we add \( x = false \) to our assignment. This distribution can bring more opportunities for unit communication.

3) The third strategy is to simply select an unassigned variable \( x \) and branch the search: try \( x = true \) on one side and \( x = false \) on the other side.
Parallelizing DPLL

I try to parallelize the Boolean guessing process. Every time a free variable is guessed, the guess is divided into two branches, a true branch and a false branch. Each branch requires additional work by guessing additional variables, and for each recursive call to the process, the work is parallelized to one thread. Although the sequential DPLL algorithm usually traverses the decision tree through depth-first search, the idea of parallelization is to traverse all paths in parallel.

```haskell
SatDPLL :: Expr -> Bool
SatDPLL expr =
  case TV expr of
    Nothing -> unC $ SPY expr'
    Just v ->
      let true1 = SPY (VarG v True expr)
          false = SPY (VarG v False expr')
      in SatDPLL true1 || SatDPLL false
    where
      expr' = litE $ Meg $ unitPropagation expr

SGI :: Strategy [Bool] -> Int -> Expr -> Bool
SGI strat depth expr =
  case TV expr of
    Nothing -> unC $ SPY expr'
    Just v ->
      let true1 = SGI strat depth' $ SPY (VarG v True expr')
          false = SGI strat depth' $ SPY (VarG v False expr')
      in or [(true1, Gfail) `using` strat]
    where
      depth' = depth - 1
      expr' = litE $ Meg $ unitPropagation expr
```

Using divide-and-conquer idea, which incrementally divides the search space into subspaces, successively allocated to sequential CDCL workers. Workers cooperate through some load-balancing strategy, which performs the dynamic transfer of subspaces to idle workers, and through the exchange of conflict clauses.

Result

I used Virtual Machining online using a 2.3 GHz Intel Xeon CPU, processor with 4 cores to generate DIMACS Benchmark Instances online settings, but found that there is no obvious and surprising difference in the running time of sequential and parallel algorithms. This is unexpected.
Runtime increasement for DPLL Parallel Algorithm Under different cores

Future Plan
I think I should try to parallelize sequential algorithms from different aspects to get better performance. (For example, simplification of expressions: Parallelizing the expression simplification led to far too great of spark creation with the vast majority being garbage collected, unit propagation: too much synchronization between threads is required.) Also, try the algorithm remotely on different machine for a fair amount of time.

Reference

Code

```haskell
module ParallelSATSolver where

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

PPS :: (NFData a) => Strategy [a]
PPS [a,b] = do
    a' <- rpar a
    b' <- rpar b
    return [a', b']
PPS _ = undefined

Depthfix :: Int
Depthfix = 50

Sat :: Expr -> Bool
```
Sat (Const b) = b
Sat orExpr@(Or _ _) = SatDPLL orExpr
Sat (And x y) = and ([SatDPLL x, SatDPLL y]
    `using` PPS)
Sat _ = undefined

SatDPLL :: Expr -> Bool
SatDPLL = SG1 PPS Depthfix

SatDPLLS :: Expr -> Bool
SatDPLLS expr =
  case FV expr' of
    Nothing -> unC $ SPY expr'
    Just v ->
      let true1 = SPY (VarG v True expr')
          Gfail = SPY (VarG v False expr')
      in SatDPLLS true1 || SatDPLLS Gfail
  where
    expr' = litE $ fNeg $ unitPropagation expr

SG1 :: Strategy [Bool] -> Int -> Expr -> Bool
SG1 _ 0 expr = SatDPLLS expr
SG1 strat depth expr =
  case FV expr' of
    Nothing -> unC $ SPY expr'
    Just v ->
      let true1 = SG1 strat depth' $ SPY (VarG v True expr')
          Gfail = SG1 strat depth' $ SPY (VarG v False expr')
      in or ([true1, Gfail] `using` strat)
  where
    depth' = depth - 1
    expr' = litE $ fNeg $ unitPropagation expr

Library.hs
module Library 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)

litE :: Expr -> Expr
litE e =
  let ls = Set.toList (lits e)
      ps = map (litP e) ls
    inP :: String -> Maybe Polarity -> Maybe (String, Bool)
exP v (Just Positive) = Just (v, True)
exP v (Just Negative) = Just (v, False)
exP _ _ = Nothing
works :: [(String, Bool)]
works = catMaybes $ zipWith exP ls ps
substitutes :: [Expr -> Expr]
substitutes = map (uncurry VarG) works
suball :: Expr -> Expr
suball = foldl (.) id substitutes
in suball e

SPY :: Expr -> Expr
SPY (Const b) = Const b
SPY (Var v) = Var v
SPY (Not expr) =
case SPY expr of
  Const b -> Const (not b)
  expr' -> Not expr'
SPY (Or x y) =
  let es = filter (/= Const False) [SPY x, SPY 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."

SPY (And x y) =
  let es = filter (/= Const True) [SPY x, SPY 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."

unC :: Expr -> Bool
unC (Const b) = b
unC _ = error "Not Const"

FV :: Expr -> Maybe String
FV (Const _) = Nothing
FV (Var v) = Just v
FV (Not e) = FV e
FV (Or x y) = FV x <|> FV y
FV (And x y) = FV x <|> FV y

VarG :: String -> Bool -> Expr -> Expr
VarG 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 = VarG var val
lits :: Expr -> Set String
lits (Var v) = Set.singleton v
lits (Not e) = lits e
lits (And x y) = Set.union (lits x) (lits y)
lits (Or x y) = Set.union (lits x) (lits y)
lits _ = Set.empty

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

litP :: Expr -> String -> Maybe Polarity
litP (Var v) v' | v == v' = Just Positive
| otherwise = Nothing

| v == v' = Just Negative
| otherwise = Nothing

fNeg :: Expr -> Expr
fNeg expr =
  case expr of
    Not (Not x) -> fNeg x
    Not (And x y) -> Or (fNeg $ Not x) (fNeg $ Not y)
    Not (Or x y) -> And (fNeg $ Not x) (fNeg $ Not y)
    Not (Const b) -> Const (not b)
    Not x -> Not (fNeg x)
    And x y -> And (fNeg x) (fNeg y)
    Or x y -> Or (fNeg x) (fNeg y)
    x -> x

litP expr v =
  case expr of
    And x y -> comP [x, y]
    Or x y -> comP [x, y]
    Not x -> error $ "Not in CNF: negation of a non-literal: " ++ show x
    Const _ -> Nothing
    _ -> error "Should never happen."

where
comP es =
  let POL = mapMaybe (flip litP v) es
  in case POL of
    [] -> Nothing
    ps -> if all (== Positive) ps
      then Just Positive
      else if all (== Negative) ps
        then Just Negative
        else Just Mixed

CLA :: Expr -> [Expr]
CLA (And x y) = CLA x ++ CLA y
CLA expr = [expr]

oneC :: Expr -> Maybe (String, Bool)
oneC (Var v) = Just (v, True)
oneC (Not (Var v)) = Just (v, False)
oneC _ = Nothing

oneP :: Expr -> Expr
oneP expr = suball expr
  where
    works :: [(String, Bool)]
    works = allCLA expr
    suball :: Expr -> Expr
    suball = foldl (.) id (map (uncurry VarG) works)

allCLA :: Expr -> [(String, Bool)]
allCLA = mapMaybe oneC . CLA

ParseIO.hs
module ParseIO where
import Library(Expr(..))
import Data.Array.Unboxed
import Language.CNF.Parse.ParseDIMACS
import Language.Haskell.Exts
CtoE :: CNF -> Expr
CtoE cnf = Cla list1
  where
    list1 = map uAtoE $ CLA cnf
    Cla :: [Expr] -> Expr
    Cla [] = Const True
    Cla [x] = x
    Cla [x,y,z] = And (And x y) z
    Cla xs = And (Cla ft) (Cla bk)
      where (ft, bk) = splitAt ((length xs + 1) `div` 2) xs

LFF :: FilePath -> IO (Either String Expr)
LFF fileName = do
cnf <- parseFile fileName
  case cnf of
    LT _ -> return $ LT "Parse Error"
    RT cnf -> return $ RT $ CtoE cnf

uAtoE :: UArray Int Int -> Expr
uAtoE = Lit . (map ItoE) . elems
  where
    ItoE :: Int -> Expr
    ItoE n | n > 0 = Var $ show n
    ItoE n = Not $ Var $ show $ abs n

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

Main.hs
module Main where
import ParallelSATSolver
import ParseIO
import System.Exit(die)
import System.Environment(getArgs, getProgName)
main :: IO ()
main = do
    param <- getArgs
    case param of
        [file] -> do
            cnf <- loadFile file
            case cnf of
                LT err -> putStrLn err
                RT cnf' -> case SatDPLL cnf' of
                    True -> putStrLn $ file ++ " is Satisfiable"
                    _    -> putStrLn $ file ++ " is UnSatisfiable"
                _    -> do
                    pn <- getProgName
                    die $ "Usage: " ++ pn ++ " <filename>"