Project Report

Parallel DPLL SAT Solver
Project Name: BoolSAT

Name:	Kenneth Kiprotich
UNI:	kk3302
Date:	12/21/2021

1 Introduction

The Boolean Satisfiability Problem involves determining whether variables of a given Boolean formula can be replaced using True or False in a way that satisfies the Boolean formula, i.e. leads the Boolean formula to evaluate to True. If the Boolean evaluates to true, it is SAT (otherwise, UNSAT). SAT is NP-Complete, making it one of the interesting problems in theoretical computer science.

For this project, I sought to explore existing SAT-solving algorithms and find ways to parallelize them in Haskell, then compare the performance of sequential approaches against the parallel one. There are several algorithms which use different heuristics. I mainly focused on the DPLL algorithm.

2 DPLL Algorithm

The DPLL (Davis-Putnam-Logemann-Loveland) Algorithm makes use of backtracking search, unit propagation, literal elimination, and recursive checks on different variable assignments to determine boolean satisfiability.

2.1 Unit Propagation

This is a step to simplify the provided CNF-SAT problem at every iteration. It involves searching for a unit clause and (1) removing instances of other clauses that contain the literal in the unit clause and (2) removing instances of the literal's complement from other clauses in the formula. This works since the problem is in Conjunctive Normal Form, so if a literal value is assigned to True, all the clauses that contain that literal value will automatically evaluate to True, and the literal's complement adds no meaning to the formula. Unit propagation reduces the search space significantly.

2.2 Literal Elimination

This is the removal of literals that do not contain any complements in the set of clauses provided. Such literals are usually called pure literals. They are eliminated from all the clauses since they can be assigned to True without having effect on the eventual satisfiability result of the formula.

2.3 Determining SAT or UNSAT

The DPLL Algorithm runs an exhaustive search on all possible assignments of variables, using unit propagation and literal elimination (discussed above) to reduce search space on different iterations. While doing the exhaustive search, the algorithm gives partial assignments to variables and determines whether taking that path will lead to satisfiability, backtracking if it does not.

A CNF formula is determined to be SAT if the variables are assigned without a clause in the CNF formula becoming empty. A CNF formula is determined to be UNSAT if the opposite happens, that is, if the assignment of variables leads to one of the clauses being empty.

2.4 DPLL Pseudocode

This is the DPLL pseudocode, adapted from here, with steps for unit-propagation and literal elimination as discussed above:

Algorithm 1 DPLL Pseudocode

```
procedure DPLL: INPUT = CNF FORMULA, OUTPUT = TRUE OF FALSE
    If Formula is a consistent set of literals, return True
    If Formula is a constains and emptey clause, return False
    For every unit clause u in the Formula:
        Formula ← unit-propagate w.r.t. u
    For every pure literal p in the Formula:
        Formula ← simplify w.r.t. p
    u ← choose a literal from Formula
    return DPLL (Formula AND u) or DPLL (Formula AND NOT u)
```

3 Implementation of DPLL in Haskell

3.1 CNF Files

A CNF file represents literals as numeric values. Negative numerals represent complements, for instance the complement of 1 is -1. Each line in the file represents one clause, and each of the lines end with a 0, which is a special value and is not treated as a literal. As expected, since it is in Conjunctive Normal Form, literals in the lines are OR-ed, while clauses across the file are AND-ed.

CNF files for testing the sequential and parallel DPLL are easily available online. Haskell has a DIMACS CNF parser library that converts the CNF formulas into a 2-dimensional Unboxed Array. I processed the clauses into a list of lists, i.e. a list of clauses.

3.2 Sequential Implementation

There are several sequential solvers easily available, including such as which uses a MiniSAT Solver available on Hackage. However, for the purpose of this project, I focused mainly on parallelizing a sequential approach that closely follows the DPLL characteristics discussed in Section 2 above.

3.3 Parallel Implementation

By studying the DPLL algorithm, we learn that the sequential approach builds up a tree-like structure for the solution space. I think that the last step is the one is especially important: a literal is chosen from the remaining clauses, and the algorithm branches into two children, with each branch assigned to a different polarity of the literal. Therefore, this part of the DPLL algorithm lends itself neatly to parallelization.

Initially, I had tried introducing parallelism to some of the subroutines in DPLL, such as unit propagation, literal elimination, or simplification of clauses. However, I concluded that these steps were not very interesting since they would not significantly narrow down the time it takes to cover the whole solution space. Unit Propagation and literal elimination, for instance, could have been parallelized using a ParList since I parse the boolean formulas into list of lists. However, I think that parallelizing tree structure was more interesting since it would significantly affect the speed at which the solution space is explored.

Below is the Haskell pseudocode for the parallelized section of the DPLL algorithm:

```
State = {[clauses] , [variable assignments]}
parDPLL: depth State
  | if formula is empty = return [assignments]
  | otherwise = do
    lit <- chooseLiteral formula
    let positive = parDPLL (i-1) (State simplified w.r.t lit)
                             (vars with lit added)
    let negative = parDPLL (i-1) (State simplified w.r.t -lit)
                             (vars with -lit added)

    if depth > 0 then
      runEval $ do
        x <- rpar falseBranch
        return (case trueBranch of
          Nothing -> x
          Just result -> return result)
    else
        case trueBranch of
          Nothing -> falseBranch
          Just result -> return result

  where
    formula = updated clauses after unitpropagation s'
    vars    = updated list of variables assignments
```

State holds the formula and a list of variable assignments. The formula is the CNF-SAT problem presented as a list of lists, where the nested lists are clauses. The formula is created by parsing a .cnf file provided as a command-line argument.

True and *False* are assigned to a literal obtained by *chooseLiteral*, and branched into two paths: trueBranch and falseBranch. This is the step that continues in parallel. The *rpar* in this case does not need an accompanying *rseq* since it evaluates fully by the time is opposite branch returns i.e. we have to check whether *positivePath* returns a *result* or *Nothing* first. Here is the *rpar* section of the algorithm which is and Eval computation performed by *runEval*:

```
x <- rpar falseBranch
  return (case trueBranch of
  Nothing -> x
  Just result -> return result)
```

The results of the parallel algorithm against the sequential one are analyzed in more details in the next section.

4 Evaluation

4.1 Datasets used for testing

I mostly worked with problems that have been determined to be unsatisfiable in order to ensure that the algorithms explore the maximum search space rather than race to find the solution. Therefore, the solution space for Unsatisfiable boolean formulas provides a better heuristic for measuring the performance of parallelism.

The files are downloadable from the SATLIB Benchmark Problems site .

4.2 Results

Below is the threadscope diagram for a parallel DPLL running on 2 cores.

I used several cnf files to test the sequential and parallel algorithms, but the results below are from one Unsatisfiable formula. The sequential algorithm took an average of 36.549 seconds over 5 runs.

These are the results from the parallel algorithm
Depth = 8, with -N2 on a 4-thread i5 computer:

Parallel Algorithm Data: depth = 8						
Depth	Total Sparks	Converted Sparks	GC'ed Sparks	Fizzled Sparks	Elapsed Time	Speedup on Sequential Time
1	1	1	0	0	18.13s	1.96
2	3	1	0	2	18.77s	1.97
4	3	1	0	2	18.57s	1.98
8	255	1	0	254	18.20s	1.96
16	65535	2	122	65411	18.06s	1.96
32	4194355	6	3989850	204499	18.19s	1.96

Depth = 8, with -N4 on a 4-thread i5 computer:

Parallel Algorithm Data: depth = 8						
Depth	Total Sparks	Converted Sparks	GC'ed Sparks	Fizzled Sparks	Elapsed Time	Speedup on Sequential Time
1	1	1	0	0	22.15	1.65
2	3	3	0	0	16.88s	2.16
4	15	5	0	10	16.80s	2.17
8	255	5	0	250	16.87s	2.17
16	65535	18	126	65391	17.29s	2.11
32	4194539	73	3982895	211571	17.03s	2.15

The results above show that parallelism on a Dual-COre Intel Core i5 Macbook. Using -N2 and -N4 arguments did not make much of a difference in runtime when compared to the sequential program. The comparison ratios between sequential runtime and parallel runtime (on both -N2 and -N4) were consistent on all the CNF Formulas.

Figure 1 below shows the eventlog diagrams obtained using threadscope using -N2 argument, with a parallelism depth of 8. The workloads on the two cores look balanced and consistent, ending at the same time. Evenlog diagrams with similar arguments (depth and number of cores) were the same as Figure 1, therefore load balancing was consistent across all the sample problems I tried.

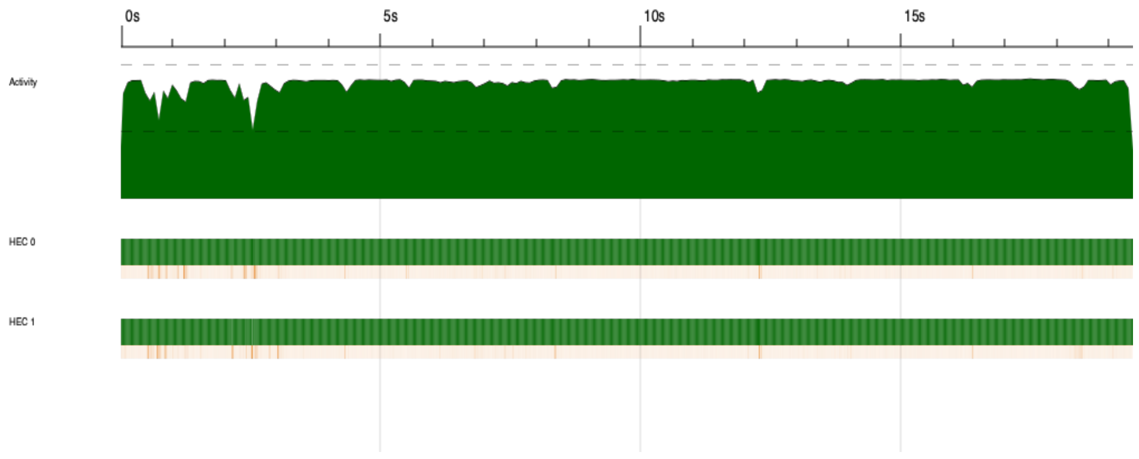

Figure 1: Eventlog for parallel DPLL: depth = 8, -N2

Figure 2 below shows the eventlog diagrams with -N4 argument and depth 8.

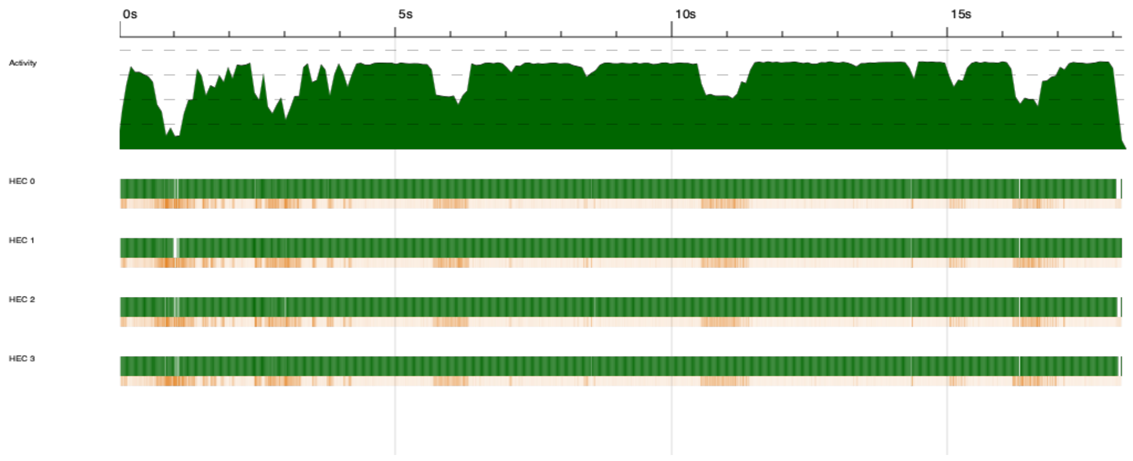

Figure 2: Eventlog for parallel DPLL: depth = 8, -N4

5 Lessons Learned

5.1 Parallelized DPLL is Faster

The parallel version of the algorithm was consistently about 2 times faster than the sequential version on all the CNF Formulas that I tested on. This is demonstrated by the table under section 4.2. The table below shows the sequential and parallel runtimes on several datasets from the SATLIB Benchmark Problems. I used parallelism depth of 8 and 2 cores (-N2) for the parallel algorithm. The Formulas with suffix *pret* are from Graph-Coloring Problems, and the ones with suffix *hole* are Pigeon-Hole Problems.

Comparison Between Parallel and Sequential Runtimes		
CNF Formula	Sequential Runtime	Parallel Time (depth 8)
pret60_25.cnf	38.574s	17.797
pret60_40.cnf	40.591s	16.884s
pret60_60.cnf	39.680s	16.85s
hole7.cnf	4.144s	1.988s
hole8.cnf	106.588s	50.30s

5.2 Unsatisfiable CNF Problems Cause A Lot of Fizzling

There was a significant difference on the ratio of sparks converted versus sparks fizzled between Satisfiable CNF Formulas and Usatisfiable CNF Formulas. As seen in section 4.2, testing on Unsatisfiable CNF Problems result in a lot of sparks getting fizzled especially at depth 8 and beyond. On the other hand, Satisfiable problems consistently resulted in more Converted Sparks than Fizzled Sparks up to depth 10.

According to Marlow (in the class text book), fizzled sparks are caused when the "expression was unevaluated at the time it was sparked but was later evaluated independently by the program." This may have occurred more in Unsatisfiable CNF Problems since parallel search on the solution space may have resulted in a branch realizing that it has finished doing the work that is still sparked for a separate branch, leading to the other work getting fizzled.

I think this is a part of the parallel algorithm that I can improve in the future.

5.3 More Cores Would Likely Increase Speedup Even More

Increasing the depth did not result in a reciprocal increase in speedup that I had expected. I ran the programs on a Dual-Core Intel Core i5 computer, which does not offer processor power to take significant advantage of Haskell parallelism's potential. I think that more cores would result in incremental speedup (up to some point) as the depth is increased.

My attempts to run on a virtual machine with more power were not successful (yet).

6 Code Listing

6.1 Lib.hs

```haskell
module Lib
    (readCNF
    ) where

import Data.Array.Unboxed
import Language.CNF.Parse.ParseDIMACS as DIMACS

readCNF :: FilePath -> IO (Either [Char] [[Integer]])
readCNF filename = do
    cnfFile <- DIMACS.parseFile filename
    case cnfFile of
        Left _ -> return $ Left "Error loading CNF file"
        Right cnfUarray -> return $ Right $ createFormula cnfUarray

createFormula :: CNF -> [[Integer]]
createFormula cnfUarray = map (map (toInteger) . elems) $ clauses
    cnfUarray
```

6.2 BoolSat.hs

```haskell
{-# LANGUAGE BangPatterns #-}
{--

ref:
https://stackoverflow.com/questions/12547160/how-does-the-dpll-algorithm-
    work
https://en.wikipedia.org/wiki/DPLL_algorithm
https://gist.github.com/adrianlshaw/1807739
https://www.cs.cmu.edu/~15414/f17/lectures/10-dpll.pdf
--}

module BoolSat where

import Data.Maybe
import Control.Parallel.Strategies

type Literal = Integer
type Clause = [Literal]
type Formula = [Clause]
type Record = [Literal]

{-
  SolverState holds the formula and the record. The formula is the CNF-SAT
  problem which is modified as different literals are assigned. The record
  holds the assignments that are 'true'
-}
data SolverState = SolverState {formula :: !Formula
                               , record :: !Record
                               } deriving (Show)

{-Sequential DPLL algorithm that uses unitpropagation and backtracking-}
seqDpll :: SolverState -> Maybe Record
seqDpll s
  | null cnf = return rec
  | otherwise = do
```

```
    l <- chooseLiteral cnf
    case seqDpll (SolverState (eliminateLiteral cnf l) (l:rec)) of
      Just res -> return res
      Nothing -> seqDpll $ SolverState (eliminateLiteral cnf (-l)) ((-l):rec)
  where
    state' = unitpropagate s
    cnf = formula state'
    rec = record state'

{-
  [DEPRECATED]: first try: parallelize dpll.
-}
trial_dpllPar :: (Ord a, Num a) => a -> SolverState -> Maybe Record
trial_dpllPar _ (SolverState [] rec) = Just rec
trial_dpllPar i s
  | null cnf = return rec
  | otherwise = do
    case getUnit cnf of
      Just u -> trial_dpllPar i $ SolverState (eliminateLiteral cnf u) (u:rec)
      Nothing ->
        let
          dlit = unwrapMaybe $ chooseLiteral cnf
          trueBranch = trial_dpllPar (i-1)
                          (SolverState (eliminateLiteral cnf dlit) (dlit:rec))
          falseBranch = trial_dpllPar (i-1)
                          (SolverState (eliminateLiteral cnf (-dlit)) ((-dlit):rec))
        in if i > 0 then
          runEval $ do
            x <- rpar falseBranch
            return (case trueBranch of
              Nothing -> x
              Just r -> return r)
           else
            case trueBranch of
              Nothing -> falseBranch
              Just r -> return r
  where
    state' = unitpropagate s
    cnf = formula state'
    rec = record state'

{-
  Parallel DPLL that does not have any depth. It parallelizes the dpll on the branching part
  i.e. the last line of the dpll pseudocode. Computes the falseBranch in par with the true one.
-}
parDpll :: SolverState -> Maybe Record
parDpll s
  | null cnf = return rec
    | otherwise = do
      l <- chooseLiteral cnf
      let trueBranch = parDpll (SolverState (eliminateLiteral cnf l) (l:rec))
      let falseBranch = parDpll (SolverState (eliminateLiteral cnf (-l)) ((-l):rec))
      runEval $ do
```

```haskell
        x <- rpar falseBranch
        return (case trueBranch of
          Nothing -> x
          Just r -> return r)

    where
      state' = unitpropagate s
    cnf = formula state'
    rec = record state'

---------------------------------
{-
  Parallel DPLL that has a depth param. It parallelizes the dpll on the
    branching part
  i.e. the last line of the dpll pseudocode. Computes the falseBranch in
    par with the true one.
-}
parDpll3 :: (Ord t, Num t) => t -> SolverState -> Maybe Record
parDpll3 i s
  | null cnf = return rec
    | otherwise = do
      l <- chooseLiteral cnf
      let trueBranch = parDpll3 (i-1) (SolverState (eliminateLiteral cnf l
    ) (l:rec))
      let falseBranch = parDpll3 (i-1) (SolverState (eliminateLiteral cnf
    (-l)) ((-l):rec))
      if i > 0 then
        runEval $ do
          x <- rpar falseBranch
          return (case trueBranch of
            Nothing -> x
            Just r -> return r)
      else
        case trueBranch of
          Nothing -> falseBranch
          Just r -> return r

    where
      state' = unitpropagate s
    cnf = formula state'
    rec = record state'
-------------------------------------

{-More info on unit propagation in the project report-}
unitpropagate :: SolverState -> SolverState
unitpropagate (SolverState cnf rec) =
  case getUnit cnf of
    Nothing -> SolverState cnf rec
    Just u -> unitpropagate $ SolverState (eliminateLiteral cnf u) (u:rec)

{-Checks for a literal in the formula and returns it (or Nothing otherwise
    )-}
chooseLiteral :: Formula -> Maybe Literal
chooseLiteral cnf = listToMaybe . concat $ cnf

{-
  Checks for a clause with only one literal and returns the literal, other
    -
  wise it returns Nothing
-}
getUnit :: Formula -> Maybe Literal
getUnit xs = listToMaybe [x | [x] <- xs]

{-
  This does what was described under the section of unit propagation in the
  report. That is, given a literal a to be simplified with, it: " (1) removes
  instances of other clauses that contain the literal and (2) removes
  instances of the literal's complement in other clauses.
  Proof: (a OR _) = a, (a OR (NOT a)) = a
 -}
eliminateLiteral :: Formula -> Literal -> Formula
eliminateLiteral cnf l = [simplClause x l | x <- cnf, not (elem l x)]
  where
    simplClause c lit = filter (/= -lit) c

{-solver for sequential; takes cnf formula as input, together with empty list
that will hold the output-}
seqDpllSolve :: [[Integer]] -> Maybe [Integer]
seqDpllSolve = seqDpll . flip SolverState []

{-solvers for parallel; takes cnf formula as input, together with empty list
that will hold the output-}
parDpllTrialSolve :: (Ord a, Num a) => a -> [[Integer]] -> Maybe [Integer]
parDpllTrialSolve i = trial_dpllPar i . flip SolverState []

parDpllSolve :: [[Integer]] -> Maybe [Integer]
parDpllSolve = parDpll . flip SolverState []


parDpllSolve3 :: (Ord a, Num a) => a -> [[Integer]] -> Maybe [Integer]
parDpllSolve3 i = parDpll3 i . flip SolverState []

{-obtaining the value in the Maybe type-}
unwrapMaybe :: Maybe a -> a
unwrapMaybe (Just n) = n
unwrapMaybe Nothing = error $ "Nothing is returned here"
```

6.3 Main.hs

```haskell
module Main where

import Lib
import BoolSat
import Control.Monad
import System.Environment(getArgs)
import System.Exit(die)

{--
  Main is set up for 3 types of tests:
--}

-- >>> 1: tests a parallel dpll that has no max
--        parallelization depth

-- main = do
--  args <- getArgs
--  case args of
--    [filename] -> do
--      cnfFormula <- readCNF filename
--      case cnfFormula of
--        Left e -> putStrLn e
--        Right formula -> putStrLn $
--                           show $
--                           parDpllSolve formula
--    _ -> putStrLn "Usage: stack run <cnf file>"


-- >>> 2: tests a parallel dpll that has a max
--        parallelization depth

main = do
  args <- getArgs
  case args of
    [filename, depth] -> do
      cnfFormula <- readCNF filename
      let d = read $ depth
      case cnfFormula of
        Left e -> putStrLn e
        Right formula -> putStrLn $
                           show $
                           parDpllSolve3 d formula
    _ -> putStrLn "Usage: stack run <cnf file> <depth of parallelism>"

-- >>> 3: tests a sequential dpll

-- main = do
--  args <- getArgs
--  case args of
--    [filename] -> do
--      cnfFormula <- readCNF filename
--      case cnfFormula of
--        Left e -> putStrLn e
--        Right formula -> putStrLn $
--                           show $
--                           seqDpllSolve formula
--    _ -> putStrLn "Usage: ./sequential <cnf file>"
```

7 References

Parallel and Concurrent Programming in Haskell by Simon Marlow
Haskell Dimacs CNF parser library on Hackage
DPLL explanation here: http://www.diag.uniroma1.it// liberato/ar/dpll/dpll.html