DPLL Algorithm

Wei Qiang

Introduction

I implemented a boolean satisfiability solver using DPLL algorithms[2] that take an input of CNF clauses, and output the model if the formula is satisfiable and output unsat otherwise. I also convert it to a paralleled version to make it faster.

The Implementation

0. CNF File format

The cnf file format is the one I found online[1]. I made a parser for this file format. The line starting with "p" is the program description of how many variables and clauses. The line starting with "c" is the comment line. For example,

p cnf 3 2
1 -3 0
2 3 -1 0

are the clauses:

(x1 ∨ (not x3) ∨ x2) ∧ (x2∨ x3 ∨ (not x1))

1. Data Types

Since we need to represent CNF clauses and each CNF clause contains several literals. Each literal can be itself or its negation. The literal can be represented as a data type with two constructor Lit String and Not String. Where Lit s represents the symbol itself, Not s represents ¬ s. A CNF clause can be represented as a type of list of literals. The symbols can be represented as a Set of strings. The model can be represented as Map of (String, Bool).

2. Core functions

```haskell
data Lit =
      Lit String
    | Not String
  deriving (Show,Eq)

type CNF = [Lit]


type Clauses = [CNF]


type Symbols = Set.Set String

type M = Map.Map String Bool
```

The core function dpll and dpll_eval are just implementations of the DPLL pseudocode, where the latter is the sequential version and dpll_eval is the parallelized version.

```haskell
dpll :: Symbols -> Clauses -> M -> Maybe M
dpll symbols cs m
    | all (isTrueInCNF m) cs = Just m
    | any (isFalseInCNF m) cs = Nothing
    | otherwise = case pures of
                    l@(x:xs) -> dpll unassigned cs new_model
                    _ -> case findUnit symbols cs m of
                           Just (s, c, m) ->
                               dpll s c m
                           Nothing ->
```

```haskell
dpll_eval2 :: Int -> Symbols -> Clauses -> M -> Maybe M
dpll_eval2 d symbols cs m
    | all (\x -> x == True) $ runEval $ parMap (isTrueInCNF m) cs = Just m
    | any (\x -> x == True) $ runEval $ parMap (isFalseInCNF m) cs = Nothing
    | otherwise = do case findPure symbols cs of
                       l@(x:xs) ->
                           let newm = foldr (\(s,b) acc -> Map.insert s b acc) m l in
                           let unassigned = foldl (\s x -> Set.delete (fst x) s) symbols l in
                                 dpll_eval2 d unassigned cs newm
                       _ -> case findUnit symbols cs m of
                               Just (s, c, m) ->
                                 dpll_eval2 d s c m
                               Nothing ->
                                   let ele = Set.elemAt 0 symbols in
                                   let truebranch = dpll_eval2 (d-1) (Set.delete ele symbols)
cs (Map.insert ele True m) in
                                   let falsebranch = dpll_eval2 (d-1) (Set.delete ele symbols)
cs (Map.insert ele False m) in
                                   if d == 0
                                   then
                                       case truebranch of
                                           Just m -> Just m
                                           Nothing -> falsebranch
                                   else
                                       runEval $ do
                                               j <- rpar $ falsebranch
                                               case truebranch of
                                                   Just m -> do return (Just m)
                                                   Nothing -> do return j
```

The dpll algorithm will find pure literals and unit clauses first, where pure literals refer to the symbols that all have the same sign and unit clauses mean that during the current model, the clause that only has one literal left unassigned and the rest are assigned false. Then, the dpll will do a simplification process that will assign the unit clauses True value and assign pure literals True if it is not negative literal, assign False otherwise. If there are no pure literals or unit clauses, then dpll will backtrack by assigning an unassigned variable both values and see if either one is True.

3. Helper methods

Unit Propagation The unit propagation is to first "unify" the clauses such that all unit clauses will be reduced to one element clause. Unit clause means that all literal except one in the clause have been set to false, the rest one is not assigned. Then, we find one unit clause and assign the literal value to True and simplify the clauses by deleting the clause that contains the unit clause literal. Then, we continue the dpll algorithm.

```
unitifyClauses :: M -> Clauses -> Clauses
unitifyClauses m clauses =
    map unifyCNF clauses
    where unifyCNF cnf = case getunitassign cnf of
                            Just (x,_) -> [x]
                            Nothing -> cnf
          getunitassign cnf = getIfone $ (filter (\(x,b) -> b) (map findmodel cnf `using` parList 
rpar) `using` parList rpar)
          findmodel l@(Lit s) = case Map.lookup s m of
                                    Just b -> (l,b)
                                    Nothing -> (l,True)
          findmodel l@(Not s) = case Map.lookup s m of
                                    Just b -> (l,not b)
                                    Nothing -> (l,True)
          getIfone [x] = Just x
          getIfone _ = Nothing
```

```
findUnit :: Symbols -> Clauses -> M -> Maybe (Symbols, Clauses, M)
findUnit s c m =
    let clauses = unitifyClauses m c in
    let unit_clauses = find (\x -> length x == 1 && isNotInM (getSymbol $ head x) m) 
clauses in
        case unit_clauses of
            Nothing -> Nothing
            Just unit ->
                let newm = Map.insert (getSymbol $ head unit) (getSign $ head unit) m in
                let symbols = Set.delete (getSymbol $ head unit) s in
                let simple_clause = simplify clauses unit in
                    Just (symbols, simple_clause, newm)
    where isNotInM symbol m =
            case Map.lookup symbol m of
            Just _ -> False
            Nothing -> True
          simplify clauses unit_clause =
            filter (\x -> not (hasunit_clause x unit_clause)) clauses
          hasunit_clause cnf unit_clause =
            case find (\x -> x == head unit_clause) cnf of
              Just _ -> True
              _ -> False
```

```haskell
findPure :: Symbols -> Clauses -> [(String,Bool)]
findPure s clause =
        posassigns ++ negassigns
    where clausesHavesymbol symbol clause = mapMaybe (ifSymbolInCNF symbol) clause
          ifSymbolInCNF symbol cnf =
           case find (\x -> getSymbol x == symbol) cnf of
             Just lit -> Just (getSign lit)
             Nothing -> Nothing
          isAllTrue sp = all (==True) sp
          isAllFalse sp = all (==False) sp
          sl = Set.toList s
          possymbols = filter (\x -> isAllTrue $ clausesHavesymbol x clause) sl
          negsymbols = filter (\x -> isAllFalse $ clausesHavesymbol x clause) sl
          posassigns = map (\x -> (x, True)) possymbols
          negassigns = map (\x -> (x, False)) negsymbols
```

Pure Literals The pure literals refer to the literals in current clauses that all have the same signs. After finding the pure literals, we assign it with True if it is not negation, otherwise assign it False.

4. Parallelism

The parallel trick I use in this project is mostly in the main algorithms. In the backtracking part, I run the evaluation of both assigning a new variable to True and to False and wait for the first result. If the first result comes to True, I just return True and don't need to wait for another result, if the first is False, then I will return the second result. I use the Strategies/Eval monad to implement the parallelism. Here I am not caring about evaluating to normal form or WHNF, so using rpar and rseq are sufficient.

```haskell
runEval $ do
          let ele = Set.elemAt 0 symbols
          i <- rpar $ dpll_eval2 (Set.delete ele symbols) cs (Map.insert ele True m)
          j <- rpar $ dpll_eval2 (Set.delete ele symbols) cs (Map.insert ele False m)
          rseq i
          case i of
            Just m -> do return (Just m)
            Nothing -> do rseq j
                          return j
```

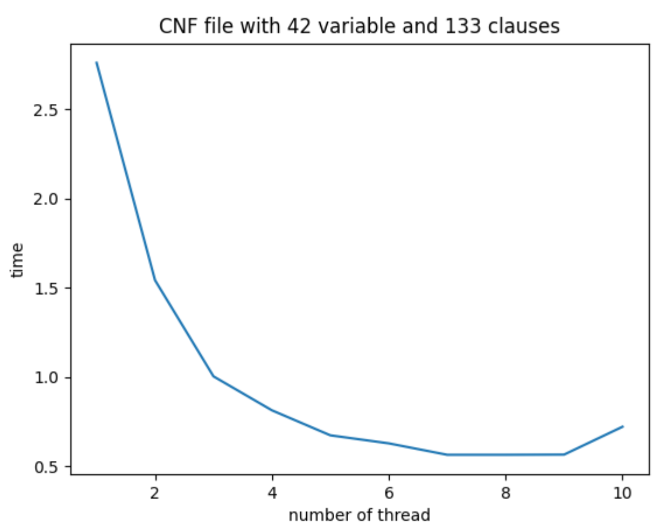
I also used Strategies in the helper functions to get list operations and map working faster.
By using parList on list operations and parMap on map operations, It becomes much faster than only using parallelism in main methods.

5. Evaluation

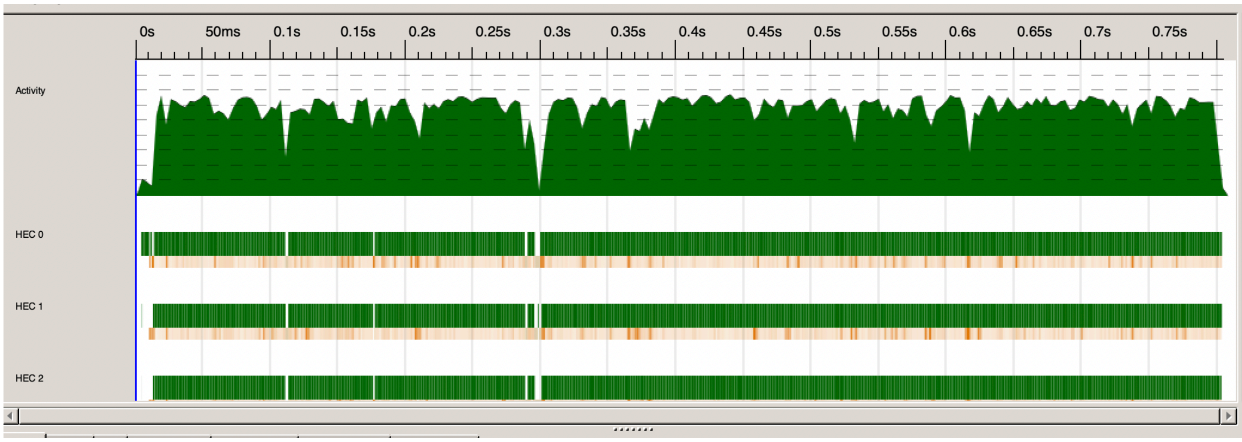
There is a substantial time difference between the parallelized version vs sequential versions as the input gets bigger. I use several tests of CNF files[1] to run in the both sequential version and parallelized version.

There are also some limitations to the testing. Since the time depends more on the difficulty of each individual problem, it is meaningless to make a plot of a function of input size vs time. (One 60-variable test can be solved in 1s, but the other might take 3 mins). But there is some pattern here, in small tests examples, the time differences are small and can be ignored; in large and hard examples, the parallel running time is generally 5-7 times faster than then sequential one.

 # threads vs time I test the speed up with a 42-variable and 133 clauses file. The plot shows that time decreases as the thread increase. At first, it decreases by nearly 50%, and then it decreases slowly as the thread increase.



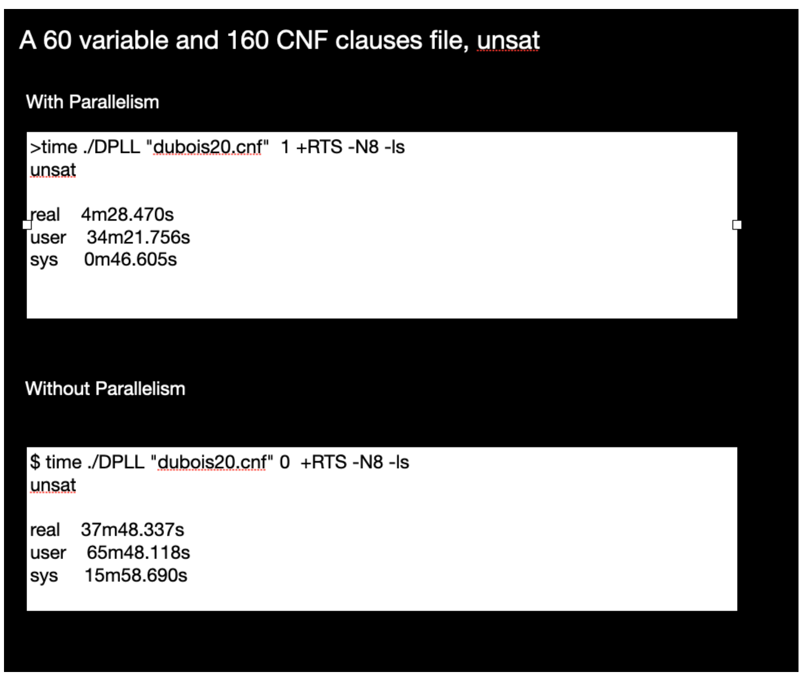
Balance The balance is pretty good for large files. For small files, they only need one thread to be solved. (they will be solved during unit propagation and pure literal finding and will not enter parallelized part).



A 50 variable, 80 clauses sat file

Testing

1) The first one is a 60-variable and 160-clause unsatisfiable CNF file[2] (in the source file). This file is a little complex cnf file and it takes 4 mins for the parallel version to output unsat but takes 37 minutes for the sequential version.



A 100 variable and 160 CNF clauses file, satisfiable

With Parallelism

```
 time ./DPLL "aim-50-1_6-yes1-4.cnf" 1 +RTS -N8 -ls
fromList [("1",False),("10",True),("11",True),("12",False),("13",False),("14",True),
("15",True),("16",True),("17",False),("18",True),("19",True),("2",True),("20",True),
("21",False),("22",True),("23",True),("24",True),("25",True),("26",False),("27",False),
("28",False),("29",False),("3",False),("30",False),("31",False),("32",True),("33",False),
("34",False),("35",True),("36",True),("37",False),("38",False),("39",True),("4",False),
("40",True),("41",False),("42",True),("43",True),("44",True),("45",False),("46",True),
("47",True),("48",False),("49",False),("5",False),("50",True),("6",True),("7",False),
("8",False),("9",False)]

real    0m0.141s
```

Without Parallelism

```
time ./DPLL "aim-50-1_6-yes1-4.cnf" 0 +RTS -N8 -ls
fromList [("1",False),("10",True),("11",True),("12",False),("13",False),("14",True),
("15",True),("16",True),("17",False),("18",True),("19",True),("2",True),("20",True),
("21",False),("22",True),("23",True),("24",True),("25",True),("26",False),("27",False),
("28",False),("29",False),("3",False),("30",False),("31",False),("32",True),("33",False),
("34",False),("35",True),("36",True),("37",False),("38",False),("39",True),("4",False),
("40",True),("41",False),("42",True),("43",True),("44",True),("45",False),("46",True),
("47",True),("48",False),("49",False),("5",False),("50",True),("6",True),("7",False),
("8",False),("9",False)]

real    0m1.131s
```

2) The second test is the easy satisfiable one but still with 100 variables and 160 clauses. It takes 0.6s for the parallel version, 4s for the sequential version. The parallel version is about 6 times faster than the sequential version.

3) The third test is also a hard 63-variable, 168 clauses unsatisfiable cnf file. It takes 9 mins to run for the parallel version, and it takes much longer(more than 1 hour) to run for the sequential version.(I canceled it since it is takes too long)

63 variables and 168 clauses, unsat

With Parallelism

```
time ./DPLL test_file/dubois21.cnf 1  +RTS -N8 -ls
unsat

real    9m13.591s
```

Without Parallelism

```
time ./DPLL test_file/dubois21.cnf 1  +RTS -N8 -ls
unsat


This one takes too long to run so I cancelled it. It is more than 1 hour.
```

6. Reference

[1] CNF files. https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html
[2] DPLL algorithms. Wikipedia. https://en.wikipedia.org/wiki/DPLL_algorithm.