A Boolean SAT solver In Haskell

DPLL Algorithm

By Wei Qiang

Boolean SAT solver

Given a set of CNF clauses, output a model if the formula is satisfiable, otherwise output unsatisfiable

$$(A \lor B) \land (\text{not } A) \land (\text{not } B \lor C)$$

A model that satisfy the formula: {A := False, B:=True, C:= True}

Formula not satisfiable: $(A \land \text{not } A)$

$\text{not } B \land (B \lor C) \land (\text{not } C \lor B)$

DPLL Algorithm

A algorithm that solve the boolean formula

```
Algorithm DPLL
    Input: A set of clauses Φ.
    Output: A truth value indicating whether Φ is satisfiable.

function DPLL(Φ)
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    if Φ is empty then
        return true;
    if Φ contains an empty clause then
        return false;
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {not(l)});
```

Data types

Literals: A or not A

```haskell
data Lit =
      Lit String
    | Not String
```

A CNF: (A ∨ B ∨ C…)

```haskell
type CNF = [Lit]
```

CNF Clauses : (A ∨ B) ∧ (B ∨ C)

```haskell
type Clauses = [CNF]
```

Model

```haskell
type M = Map.Map String Bool
```

Core Implementation

```haskell
dpll_eval :: Int -> Symbols -> Clauses -> M -> Maybe M
dpll_eval 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 =  case pures of
                        l@(x:xs) ->
                            let newm = foldr (\(s,b) acc -> Map.insert s b acc) m pures in
                            let unassigned = foldl (\s x -> Set.delete (fst x) s) symbols pures in
                                dpll_eval d unassigned cs newm
                        _ -> case findUnit symbols cs m of
                                Just (s, c, m) ->
                                  dpll_eval d s c m
                                Nothing ->
                                    let ele = Set.elemAt 0 symbols in
                                    let truebranch = dpll_eval (d-1) (Set.delete ele symbols) cs (Map.insert ele True m) in
                                    let falsebranch = dpll_eval (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 rseq j
                                                                  return j
    where pures = findPure symbols cs
```

The project

./DPLL <filename> <1-paralial, 0-normal> +RTS -N8 -ls

The program takes a cnf file with several clauses and an int indicating running in parallel or not.

The program output a list of sufficient values of variables if formula is satisfiable
Otherwise output “unsat”

CNF File Format

```
c comments comments
p cnf <number of variable> <number of clauses>
1 2 3 0
2 3 -4 0
```

$$(X1 \lor X2 \lor X3) \land (X2 \lor X3 \lor \text{not } X4)$$

Tests

A 60 variable and 160 CNF clauses file, unsat, 9 times faster

With Parallelism

```
>time ./DPLL "dubois20.cnf"  1 +RTS -N8 -ls
unsat

real    4m28.470s
user    34m21.756s
sys     0m46.605s
```

Without Parallelism

```
$ time ./DPLL "dubois20.cnf" 0  +RTS -N8 -ls
unsat

real    37m48.337s
user    65m48.118s
sys     15m58.690s
```

Tests

A 50 variable and 80 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
```

Tests

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.
```

Tests

66 variables and 176 clauses, unsat

With Parallelism

```
time ./DPLL ./test_file/dubois22.cnf 1  +RTS -N8 -ls
unsat

real    19m51.033s
```

Without Parallelism

```
time ./DPLL test_file/dubois22.cnf 0  +RTS -N8 -ls


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

Tests

42 variables and 133 clauses, unsat

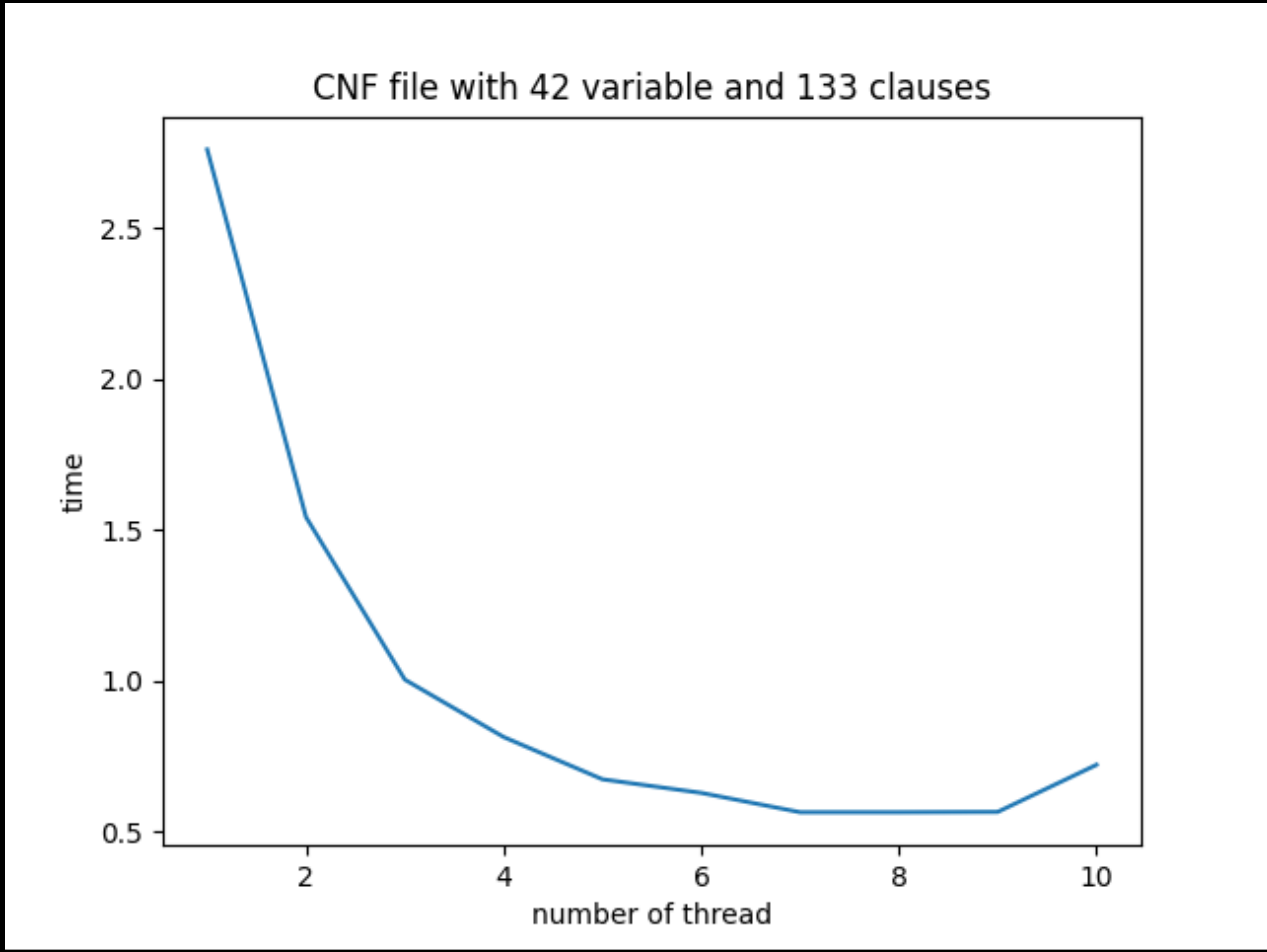
With Parallelism

```
time ./DPLL ./test_file/hole6.cnf 1  +RTS -N8 -ls
unsat

real    0m0.666s
```

Without Parallelism

```
time ./DPLL ./test_file/hole6.cnf 0  +RTS -N8 -ls
unsat

real    0m4.080s
```

hole6.cnf, 42 variable, 133 clauses, number of thread vs time

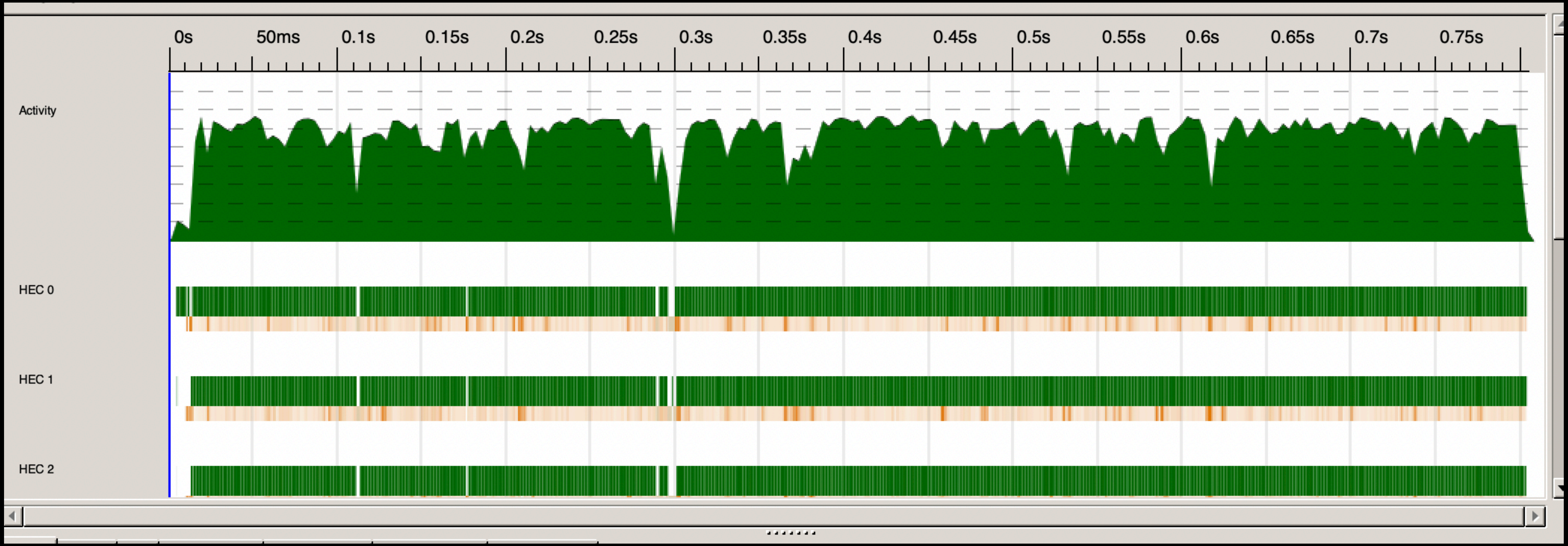


The test machine is MacBook Pro M1, 10 Cores

In small tests examples, the time difference is small and can be ignored.

In large and complex CNF files,
The parallel running time is generally 8-10 times faster than the sequential one across all test cases

The running time is very dependent on each individual problem, so it is hard to find a relationship between input size and time. So it is best to just compare each individual problem running time sequentially and in parallel.


0s
50ms
0.1s
0.15s
0.2s
0.25s
0.3s
0.35s
0.4s
0.45s
0.5s
0.55s
0.6s
0.65s
0.7s
0.75s
Activity
HEC 0
HEC 1
HEC 2

Reference

DPLL Algorithms. https://en.wikipedia.org/wiki/DPLL_algorithm