Parallel CNF-SAT Solver

Kevin Durand (kpd2136), Jonathan Tavarez (jt3481), Max Hahn (meh2280)

1 Introduction

1.1 Project Overview

This project is a parallel CNF-SAT solver which implements the Davis-Putnam-Logemann-Loveland (DPLL) recursive backtracking algorithm. It takes a DIMACS CNF file as input and returns "SAT" along with accompanying assignments, or "UNSAT".

1.2 The Boolean Satisfiability Problem (SAT)

The Boolean Satisfiability Problem (SAT) is the problem of determining whether a propositional Boolean formula can be made true by assigning true or false values to its variables [1]. It is one of the most studied problems in computer science, with both theoretical and practical significance.

SAT is theoretically significant because it is the first problem proved **NP**-complete, and therefore currently known to have exponential worst case complexity (unless $\mathbf{P} = \mathbf{NP}$). It is also practically relevant in areas such as artificial intelligence, software engineering, operations research, bioinformatics, game theory, cybersecurity, physics, and more.

Although SAT is **NP**-complete, many SAT instances from real world applications with thousands of variables can be solved in minutes or even seconds. It is believed this is because these instances contain hidden structures, "backdoors" and "backbones" to name a few. On the other hand, randomly generated instances can be quite difficult for SAT solvers. They possess a "Phase Transition Phenomenon" property: the hardness of random instances increases as the ratio of the number of clauses to the number of variables increases, up to a maximum, then decreases thereafter. For 3-SAT, this threshold is 4.26.

Because of this significance, many types of algorithms for SAT solving have been proposed. Broadly, they fall into the incomplete algorithm and complete algorithm categories. We concern ourselves with the latter category, in which algorithms are guaranteed to provide a correct answer of (un)satisfiability for a given input instance. Often, they are based on the following approaches: existential quantification, inference rules, search, and a search with inference. Our implementation of the DPLL algorithm falls under the search approach. However, it should be noted that most modern solvers utilize the last approach, with DPLL at its core [4].

1.3 DIMACS CNF

SAT solvers typically accept propositional formulas in conjunctive normal form (CNF). A CNF formula consists of a conjunction of clauses, each clause consisting of a disjunction of literals, and each literal being either a variable's positive occurrence or its negative occurrence. In other words, an AND of ORs.

$$(x_1 \wedge x_2 \wedge x_3) \wedge (\neg x_1 \wedge x_2) \wedge (\neg x_2 \wedge x_3)$$

The above example has 3 variables x_1, x_2, x_3. $\neg$ means negation (logical NOT), $\wedge$ means disjunction (logical OR), and $\wedge$ means conjunction (logical AND). This example is satisfiable because there exists an assignment of values $x_1 = \mathit{true}, x_2 = \mathit{true}, x_3 = \mathit{true}$, for example, which makes the formula true:

$$\begin{aligned}(\top \wedge \top \wedge \top) \wedge (\neg\top \wedge \top) \wedge (\neg\top \wedge \top) &\leftrightarrow \\ \top \wedge \top \wedge \top &\leftrightarrow \\ \top &\end{aligned}$$

DIMACS CNF files are textual representations of CNF formulas. Any line that begins with the character `c` is a comment line. Depending on the variation, they may be anywhere in the file, but are most commonly found at the top. After any comment lines, there must be a problem line of the form: `p cnf <variables> <clauses>`, where `<variables>` and `<clauses>` denote the number of variables and the number of clauses in the formula, respectively. Following the problem line are clause lines, which are space separated literals in the form of positive integers, delimited by a `0`. The example CNF formula above, ignoring the comment lines, would be represented as:

```
c Example DIMACS CNF input file
c Expected:  SAT
p cnf 3 3
1 2 3 0
-1 2 0
-2 3 0
```

These DIMACS CNF files are inputs to our sequential and parallel DPLL implementations. The majority of them belong to the SATLIB benchmark problems [3]. Having contextualized SAT and DIMACS CNF format, an explanation of the DPLL algorithm follows.

1.4 Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Proposed in the early 1960s, this recursive backtracking algorithm forms the basis of most modern SAT solvers. Given a CNF formula, the algorithm chooses a variable according to a heuristic and assigns a value (true or false), then simplifies the formula and recursively checks whether the simplified formula is satisfiable – if this is the case, the original formula is satisfiable; otherwise, the same recursive check is performed with the opposite assignment. This is known as the "splitting rule", since at each recursive level the formula is simplified into two smaller sub-formulas [2]. Furthermore, because at each level a variable is chosen according to some heuristic, the chosen variable is referred to as the "branching variable". The heuristic itself is referred to as the "branching heuristic", and effectively renders DPLL as a family of algorithms. Efficiency heavily depends on the choice of heuristic.

In addition to the splitting rule at each step are the Unit Propagation (also known as Boolean Constraint Propagation, or BCP for short) and Pure Literal Elimination rules. If a clause is a "unit clause", i.e. it contains only one literal, the clause can only be satisfied by assigning the necessary value to make this literal true. Thus, it may be removed from the formula, as well as every clause containing its complement. Unit Propagation often results in deterministic cascades of units, eliminating a large part of the naive search space.

If a variable occurs with only one polarity in the formula, i.e. occurs only as a positive literal x or only as a negative literal $\neg x$, it is "pure". Pure literals can be assigned a value such that all clauses containing it become true. Thus, clauses containing pure literals may also be removed from the formula.

Unsatisfiability given a partial assignment occurs when a clause becomes empty, i.e. all of its variables have been assigned values such that their corresponding literals become false. Satisfiability of the original given formula occurs when either all variables are assigned without creating an empty clause, or, in modern implementations, all clauses are satisfied. Unsatisfiability of the original given formula only occurs after exhaustive search.

2 Sequential Implementation

2.1 Zeroth Attempt: Brute Force Enumeration

x_1	x_2	$(x_1 \wedge x_2)$
false	false	false
false	true	false
true	false	false
true	true	true

To better understand the scope of the problem and verify that we could correctly parse a DIMACS CNF file, we implemented a naive algorithm in which we simply generated all possible configurations (2^n, where n is the number of variables). For each configuration, we checked if it satisfied the input formula. If none of them satisfied the formula, the formula is unsatisfiable, so we returned "UNSAT"; otherwise, the formula is satisfiable, so we returned "SAT" and the first configuration that satisfied the formula.

2.2 First Attempt: Sequential DPLL With Binary Encodings

A better approach to this problem would instead apply the Unit Propagation and Pure Literal Elimination techniques described in Section 1.4. Our initial thought was to convert the numbers, which represent variables, into a binary encoding of trues (1s) and falses (0s). With this new representation of a formula, we would then apply DPLL on grouped encodings of the same lower bits, since all variable encodings share the same subset. For example, setting the lower two bits of a number to be equal generates 4 equivalence classes:

- 0, 4, 8 all end in 00 or false, false
- 1, 5, 9 all end in 01 or false, true
- 2, 6, 10 all end in 10 or true, false
- 3, 7, 11 all end in 11 or true, true

This did not work well. 20 variable problems took ~ 1 second to be solved and 50 variable problems took > 2 minutes. The reason for poor performance was because we had to check every configuration within each equivalence class to determine if the resulting encoding was (un)satisfiable, despite simultaneously setting multiple variable assignments and implementing Unit Propagation and Pure Literal Elimination. Further work needed to be done to optimize formula rearrangement to some acceptable extent, which at the time seemed less desirable than simply adopting a different representation.

2.3 Second Attempt: Haskell Optimizations

Instead of using binary representations, we represent formulas as `[Clauses]` and clauses as `[Literals]`. A `Literal` is simply identified by an `Int` parameter. This internal representation of the formula is then kept alongside a continually updated `Vector (Maybe Bool)`, where the value at a given index i represents an (un)assigned value of `Nothing`, `Just True`, or `Just False` for variable x_{i+1}.

Reading in DIMACS CNF input files then boils down to `String`-to-`Int` conversion, and formula simplification becomes constructing smaller and smaller 2D `Int` arrays. Trying out different assignments is effectively reconstructing a `Vector (Maybe Bool)` which is $O(n)$ in the number of variables, since vectors are immutable. We intended to refactor using a mutable vector, but ran out of time to do so. Other data structures we considered refactoring with include adjacency lists, the Head-Tail data structure, and Watched Literals (along with any supporting auxiliary data structures like "watches" and a "trail"). Notably, however, the performance of the sequential implementation was quite impressive, solving a randomly generated 3-SAT satisfiable instance with 150 variables and 645 clauses, i.e. with near-peak hardness of 4.3, in 38 seconds, even with a naive branching heuristic. We simply selected the first unassigned variable in the `Vector (Maybe Bool)`. A randomly generated 3-SAT unsatisfiable instance with 125 variables and 538 clauses, i.e. also with near-peak hardness of 4.3, solved in 14 seconds.

3 Parallel Implementation

3.1 First Attempt: True/False Assignments

The first intuitive choice was to parallelize the true and false guesses for each variable. We branched down both the true and false paths of the search space at any given depth level using `par` and `parseq`, sparking once for the false assignment and using `parseq` to evaluate the second assignment on the current spark. Using this implementation, we ran into various issues. Mainly, sparks were becoming "duds", indicating they weren't being efficiently used for parallel computations. After trying several different `par-parseq` combinations, we decided to try a different approach.

3.2 Second Attempt: True/False Assignments with Depth Limiting

Our second approach incorporated depth-limiting with the `parPair` strategy from lecture. This `Strategy` helped optimize the evaluation of both true and false assignments simultaneously, and depth-limiting helped curtail any over-eager parallelism ("dudding"). Out of concern for time and because preliminary results were promising, we decided to stick with this implementation for testing.

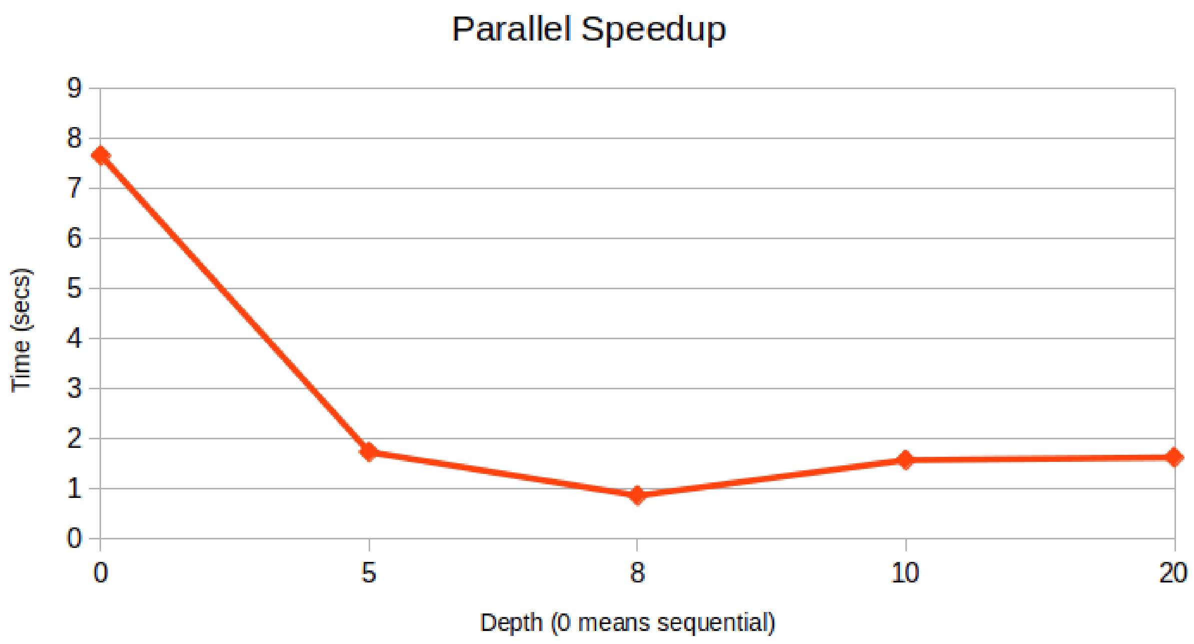
3.3 Third Attempt: Top-Level k-Split

Our third attempt introduced top-level guided k-splitting with a frequency heuristic. 2^k disjoint subproblems were generated, where each subproblem represented a specific combination of true/false assignments for the k most frequent variables in the input formula. These disjoint subproblems were then evaluated in parallel. Within each subproblem, the previous depth-limited parallel DPLL explored the remaining search space by simultaneously evaluating true/false for the current variable under consideration at a specific depth, selected naively just as before.

If time allowed, the next attempt might intuitively be implementing k-splitting at each level of recursion. Assigning multiple variables at once would reduce the depth of the search space tree while increasing the branching factor. This would potentially better-distribute subproblems across threads, especially for highly imbalanced instances where some branches terminate earlier than others, in which case splitting would be effectively performing early pruning. It would also increase the granularity of parallel tasks, lending to better utilization of threads. However, the number of subproblems generated grow exponentially in k, leading to diminishing returns in the face of mounting memory overhead from formula copies, scheduling overhead from thread saturation, increased load balancing complexity, and potentially computationally demanding or worst-case k-splits. Additional considerations might then be depth-limiting k-splitting, dynamic reduction in k based on recursion depth, and selective k-splitting according to a certain condition like polarity imbalance of remaining variables.

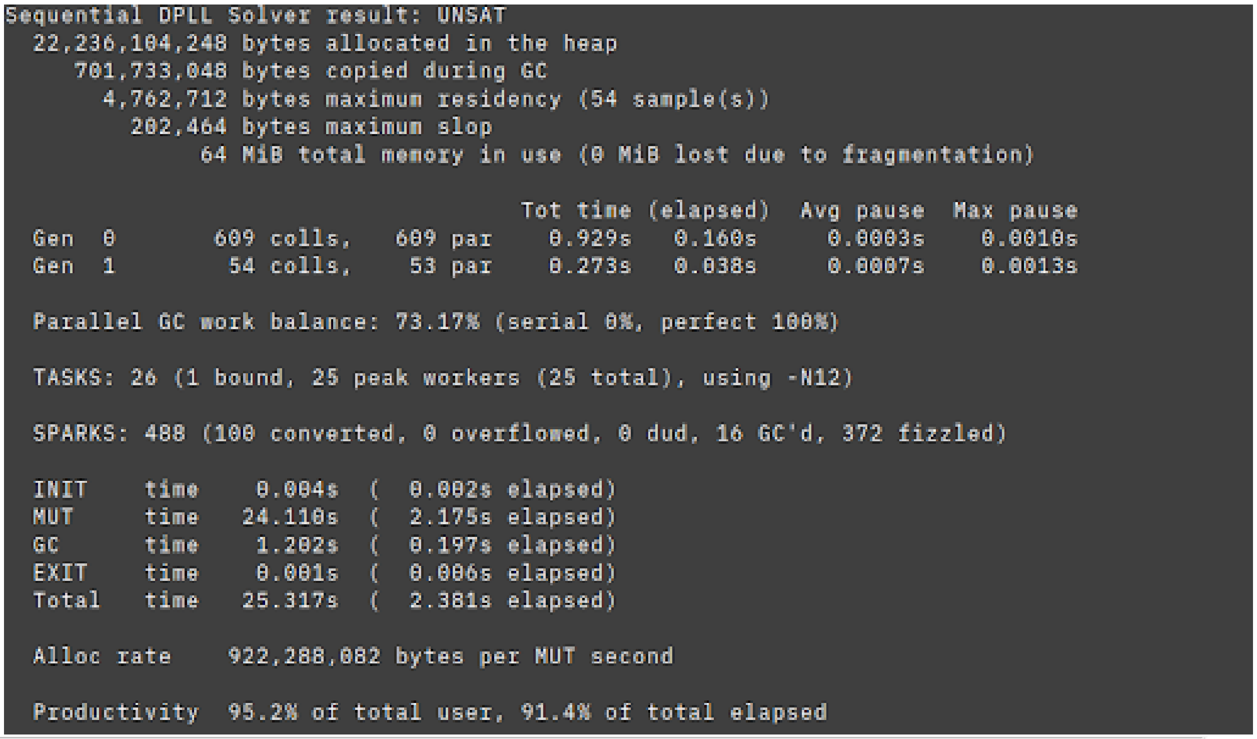
4 Testing

The following graph was constructed using 5 samples from `nuf100-430/`. Each instance contains 100 variables and 430 clauses. We can observe a speedup using all threads at different depths.

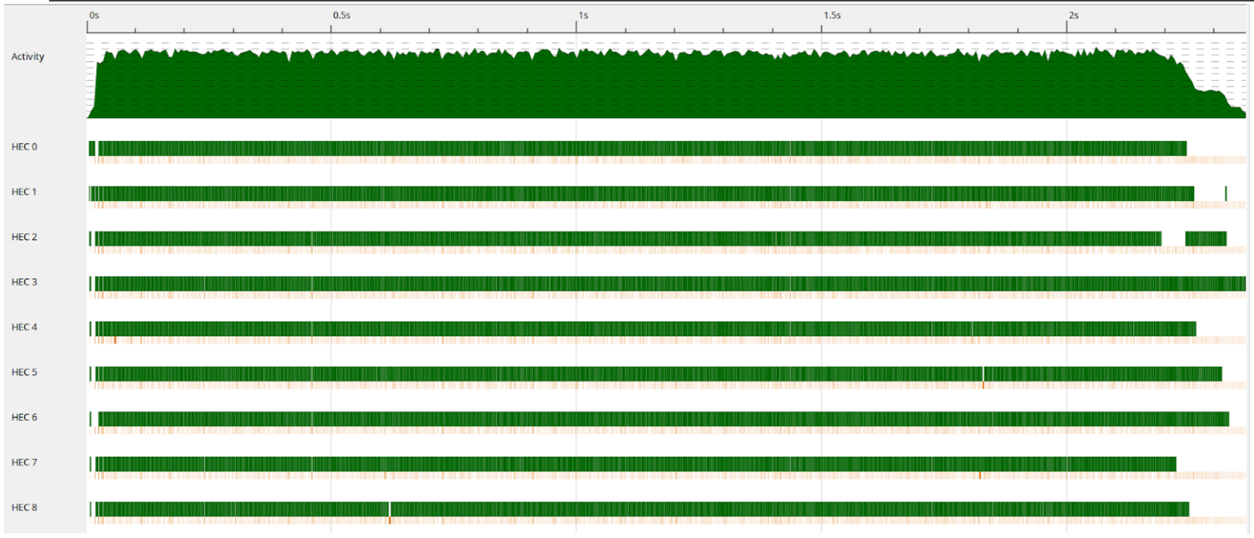


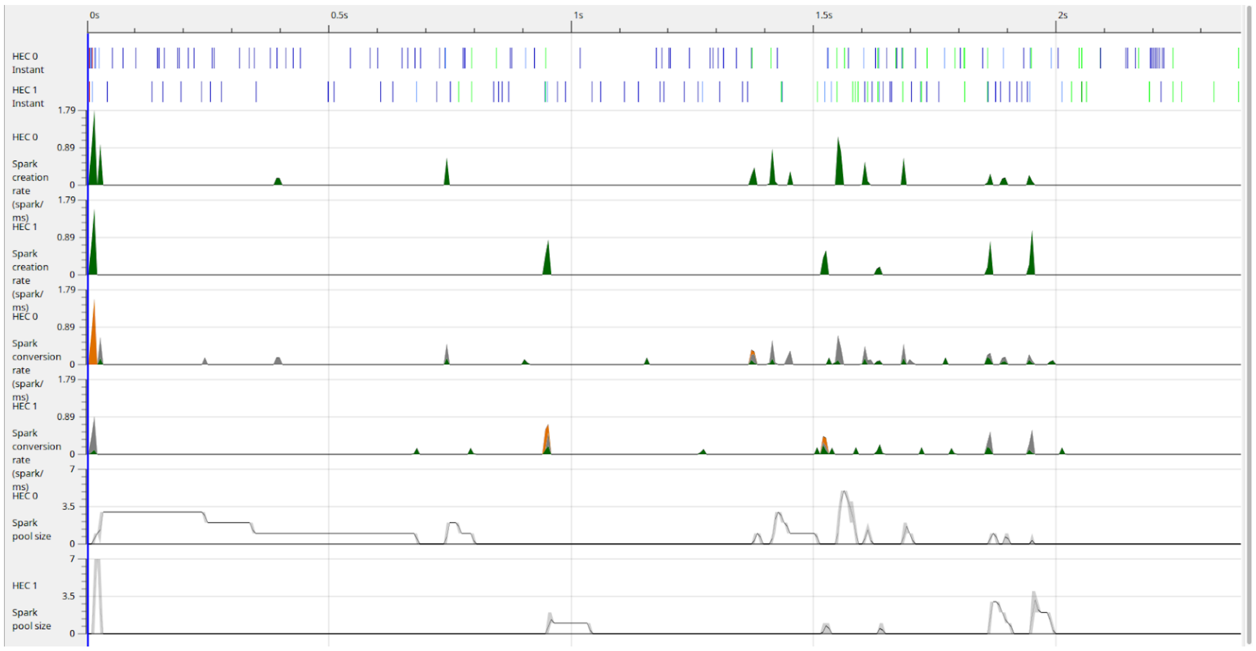
As we can see, the best results come from a depth of d=8.

	sequential	parallel, d=5	parallel, d=8	parallel, d=10	parallel, d=20
avg	7.652	1.736	1.444	1.577	1.628
std. dev.	3.644	.788	.626	.643	.648



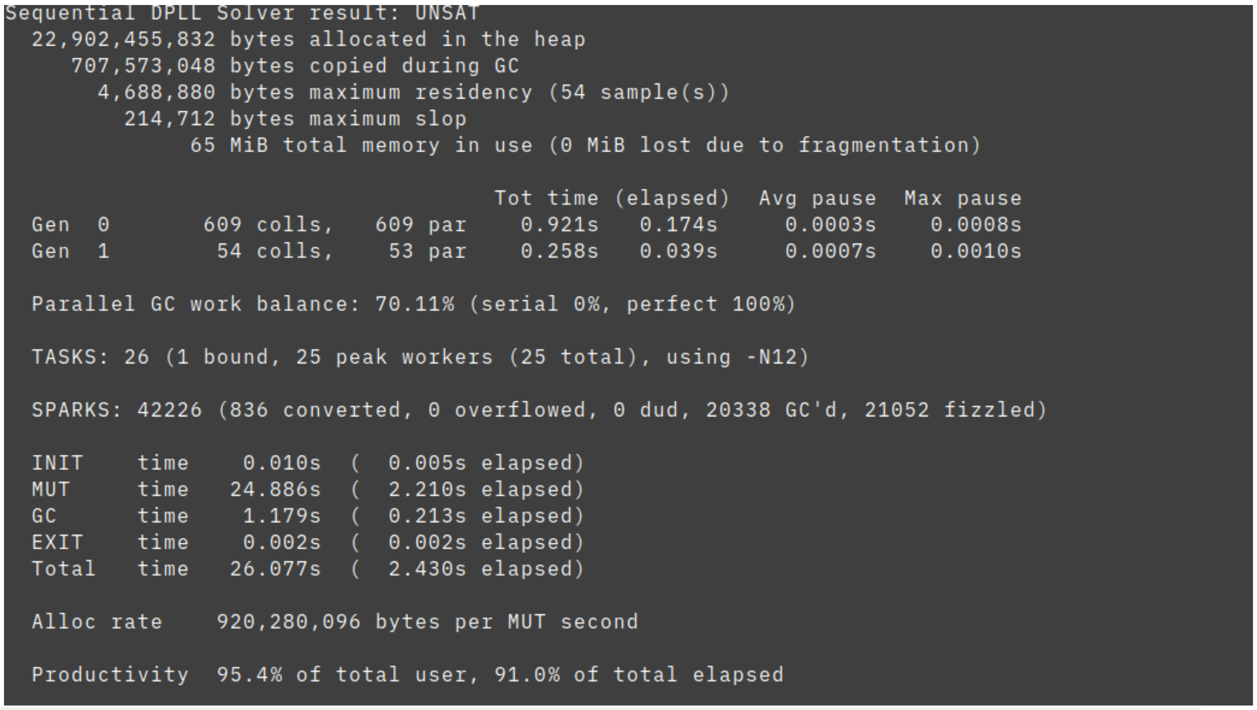
```
Sequential DPLL Solver result: UNSAT
  22,236,104,248 bytes allocated in the heap
     701,733,048 bytes copied during GC
       4,762,712 bytes maximum residency (54 sample(s))
         202,464 bytes maximum slop
              64 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       609 colls,   609 par    0.929s   0.160s     0.0003s    0.0010s
  Gen  1        54 colls,    53 par    0.273s   0.038s     0.0007s    0.0013s

  Parallel GC work balance: 73.17% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 488 (100 converted, 0 overflowed, 0 dud, 16 GC'd, 372 fizzled)

  INIT    time    0.004s  (  0.002s elapsed)
  MUT     time   24.110s  (  2.175s elapsed)
  GC      time    1.202s  (  0.197s elapsed)
  EXIT    time    0.001s  (  0.006s elapsed)
  Total   time   25.317s  (  2.381s elapsed)

  Alloc rate    922,288,082 bytes per MUT second

  Productivity  95.2% of total user, 91.4% of total elapsed
```



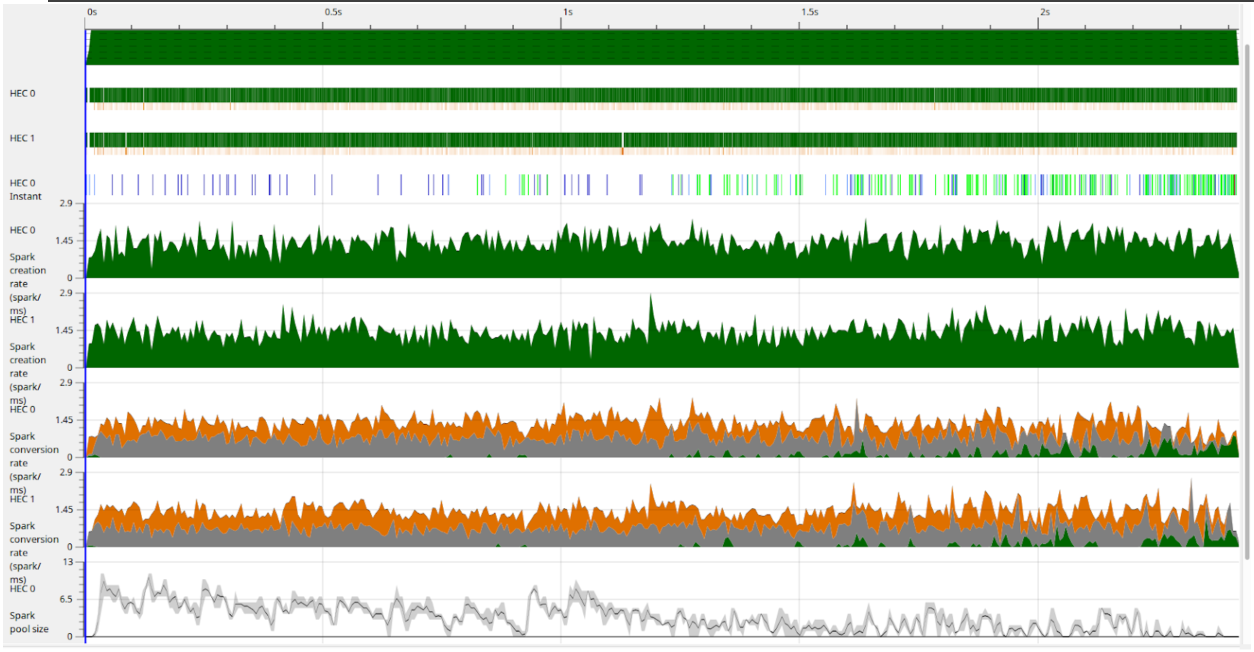


These threadscope images show spark activity with the optimal depth for the aforementioned problems. As we can see, many sparks fizzle but a lot of parallel work was done. This is might be because our sequential portion consists of assigned variable tracking with a `Vector`, Unit Propagation, and Pure Literal Elimination. These actions alone are not very computationally expensive, so on branches where a result is quickly found, evaluation of the opposite assignment may be called too quickly, resulting in the spark not being able to be run in parallel and subsequently “fizzling”. From an algorithmic standpoint, we figured not much could be done to reduce this, since quick yet inexpensive formula reduction is the core method to find a potentially satisfiable configuration. On the other hand, waiting for stable results led to all sparks becoming “duds”.

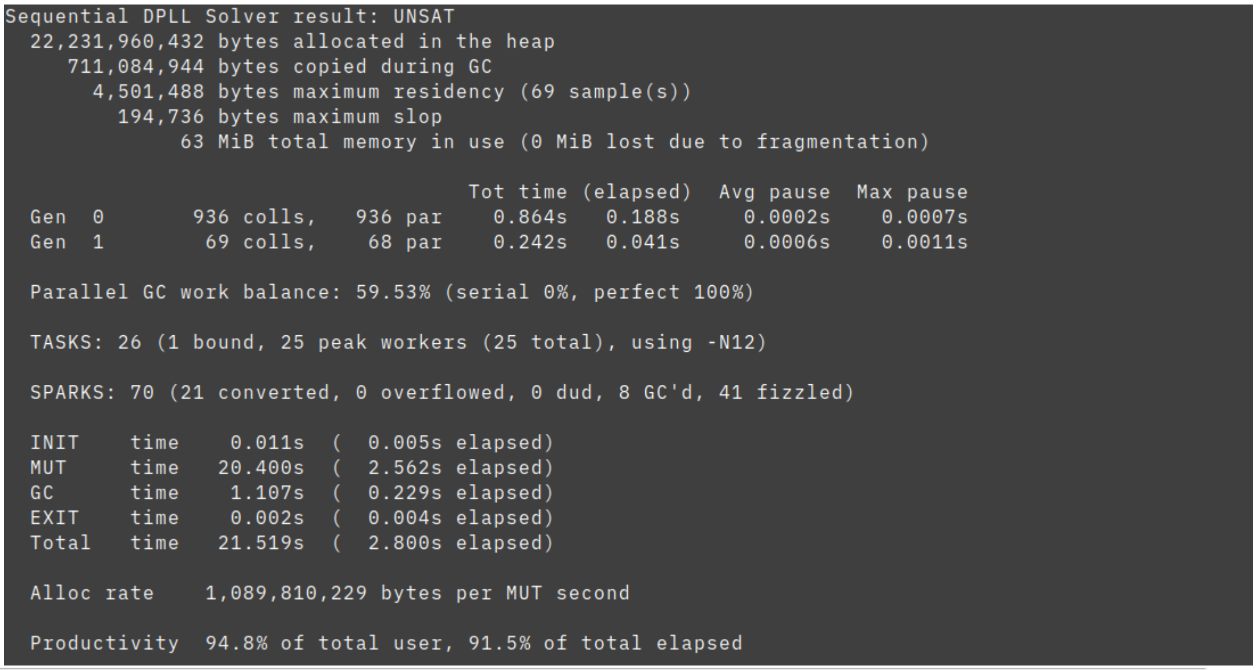
Our other hypothesis is that the high degree of “fizzling” is due to redundant/unnecessary traversals of sections of the search tree. If a branch (spark) produces results or constraints that render another branch redundant, e.g. the solution is already found, or the branch is infeasible, the other spark may fizzle because its result is no longer needed. In other words, depth-limiting mitigated, but did not resolve a core issue of over-sparking for both true/false assignments. One remedy would be to implement global shared state or hashing to track visited or resolved sections of the search space. Commonly, this is done with conflict driven clause learning (CDCL). Other more general remedies would be to implement dynamic load balancing, smarter sparking decisions, e.g. only for unexplored, promising branches, and coarser granularity control. To these ends, our initial attempt at using `par` and `parseq` might be revisited.



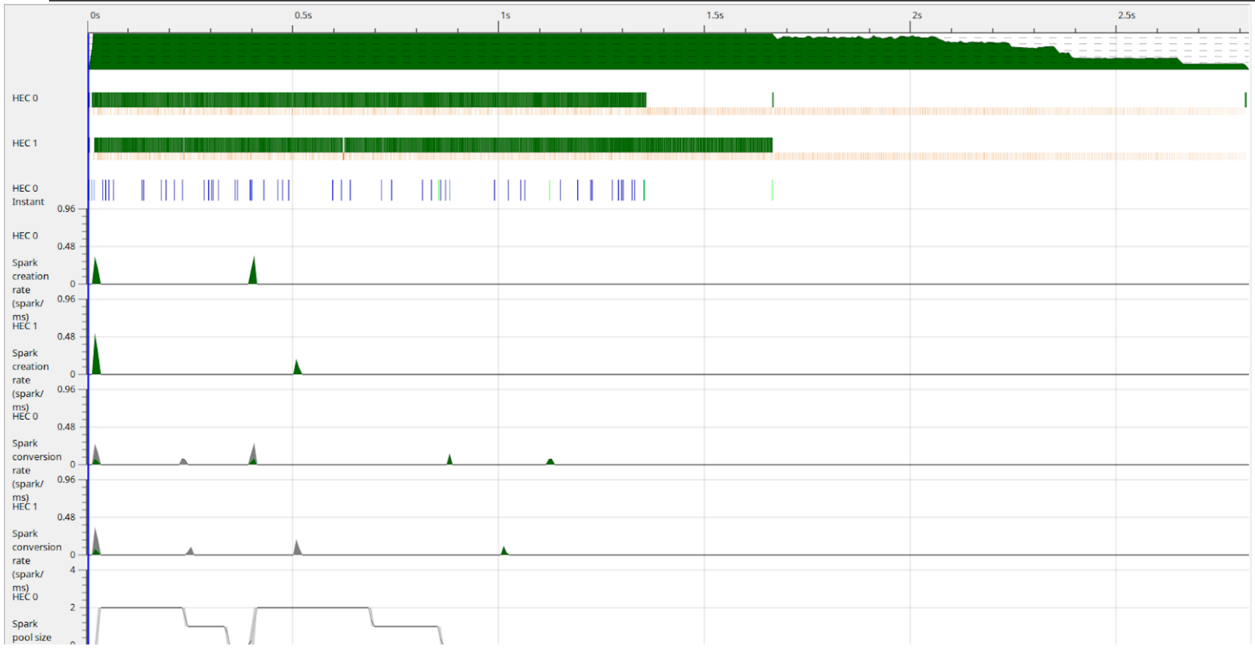
```
Sequential DPLL Solver result: UNSAT
  22,902,455,832 bytes allocated in the heap
     707,573,048 bytes copied during GC
       4,688,880 bytes maximum residency (54 sample(s))
         214,712 bytes maximum slop
              65 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       609 colls,   609 par    0.921s   0.174s     0.0003s    0.0008s
  Gen  1        54 colls,    53 par    0.258s   0.039s     0.0007s    0.0010s

  Parallel GC work balance: 70.11% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 42226 (836 converted, 0 overflowed, 0 dud, 20338 GC'd, 21052 fizzled)

  INIT    time    0.010s  (  0.005s elapsed)
  MUT     time   24.886s  (  2.210s elapsed)
  GC      time    1.179s  (  0.213s elapsed)
  EXIT    time    0.002s  (  0.002s elapsed)
  Total   time   26.077s  (  2.430s elapsed)

  Alloc rate    920,280,096 bytes per MUT second

  Productivity  95.4% of total user, 91.0% of total elapsed
```



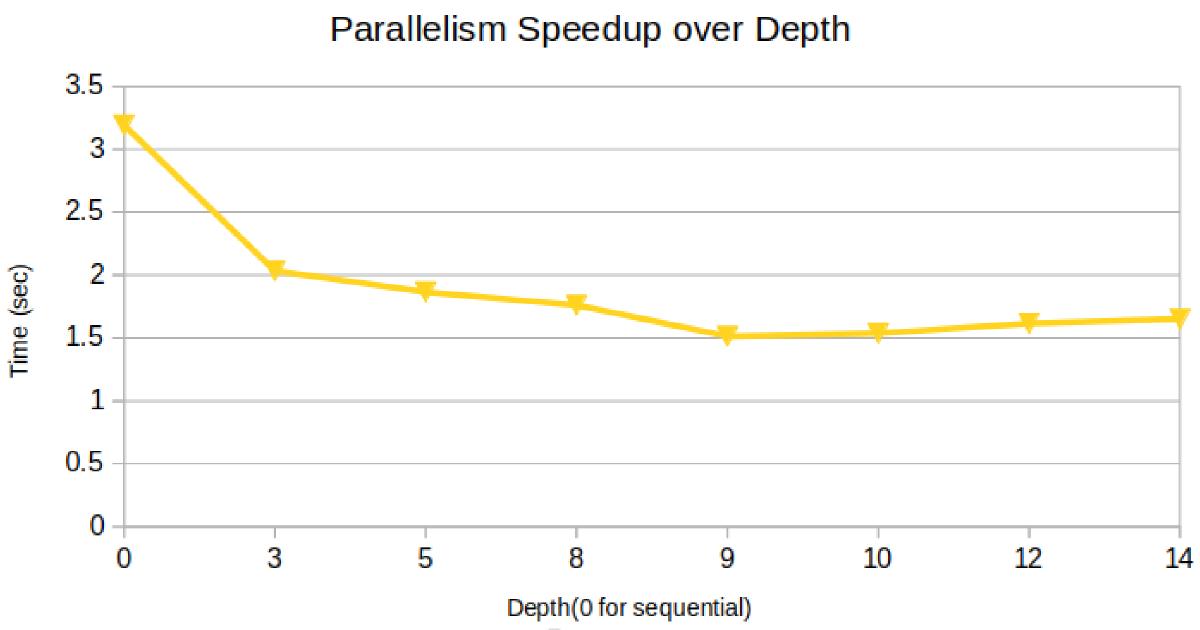
These threadscope images are for our experiments using a depth of 20. A lot more sparks are GC'd or fizzled. Both previous hypotheses can be extended to conclude this may indicate over-sparking, where spawning and garbage collecting unused sparks negatively increased runtime.

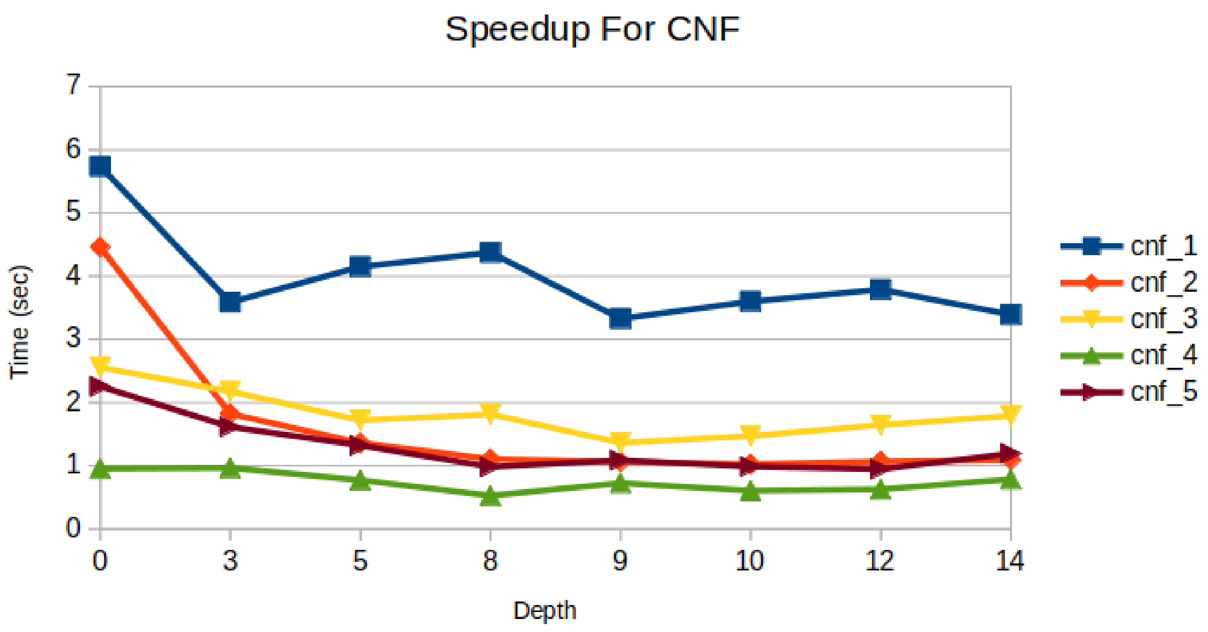


```
Sequential DPLL Solver result: UNSAT
  22,231,960,432 bytes allocated in the heap
     711,084,944 bytes copied during GC
       4,501,488 bytes maximum residency (69 sample(s))
         194,736 bytes maximum slop
              63 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       936 colls,   936 par    0.864s   0.188s     0.0002s    0.0007s
  Gen  1        69 colls,    68 par    0.242s   0.041s     0.0006s    0.0011s

  Parallel GC work balance: 59.53% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 70 (21 converted, 0 overflowed, 0 dud, 8 GC'd, 41 fizzled)

  INIT    time    0.011s  (  0.005s elapsed)
  MUT     time   20.400s  (  2.562s elapsed)
  GC      time    1.107s  (  0.229s elapsed)
  EXIT    time    0.002s  (  0.004s elapsed)
  Total   time   21.519s  (  2.800s elapsed)

  Alloc rate    1,089,810,229 bytes per MUT second

  Productivity  94.8% of total user, 91.5% of total elapsed
```

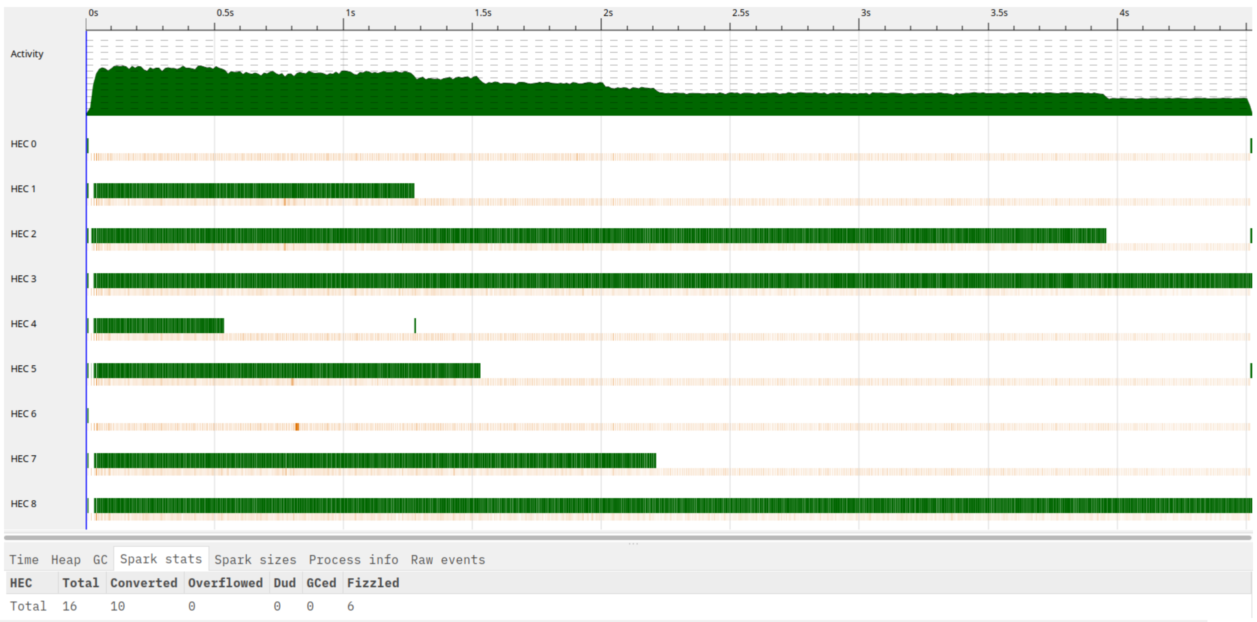


These threadscope images are for our experiments using a depth of 5. As we can see, not enough sparks were created, so a lot of work ended up being done sequentially rather than in parallel. Although some sparks fizzled, this may be due to the fact that some branches terminate/return faster than the creation of new sparks, resulting in a forced evaluation of the opposite branch before the corresponding thread starts or completes this work in parallel.

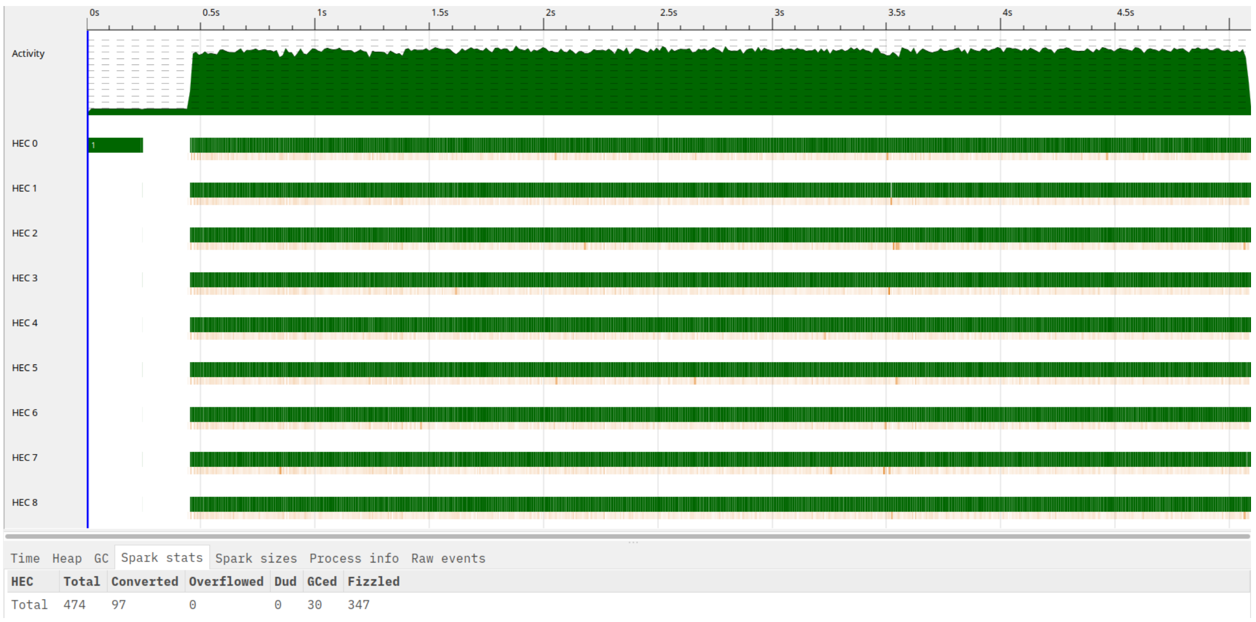




These graphs are the results of testing using the `uf100-430/` files. On average, a depth of 9 caused the most incremental speedup, but for the first instance this was achieved with a depth of 3, followed by a depth of 8. The speedup for different instances were not consistent with one another, which might explained by the manner in which our branching system works. Over-sparking may be occurring at different points for structurally different instances.



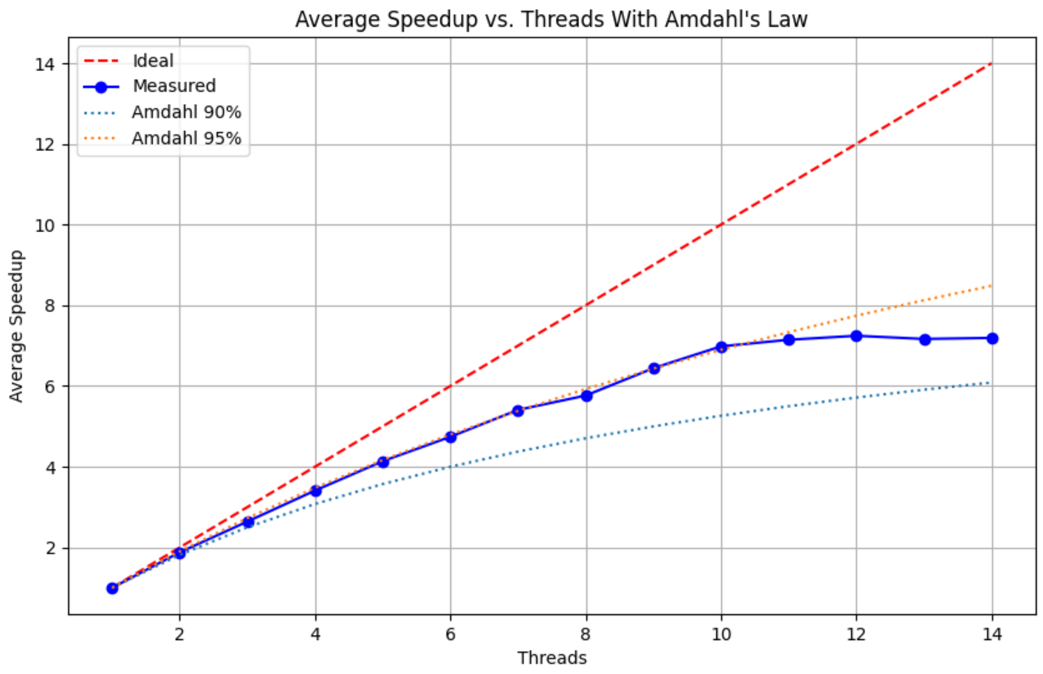
HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	16	10	0	0	0	6



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	474	97	0	0	30	347

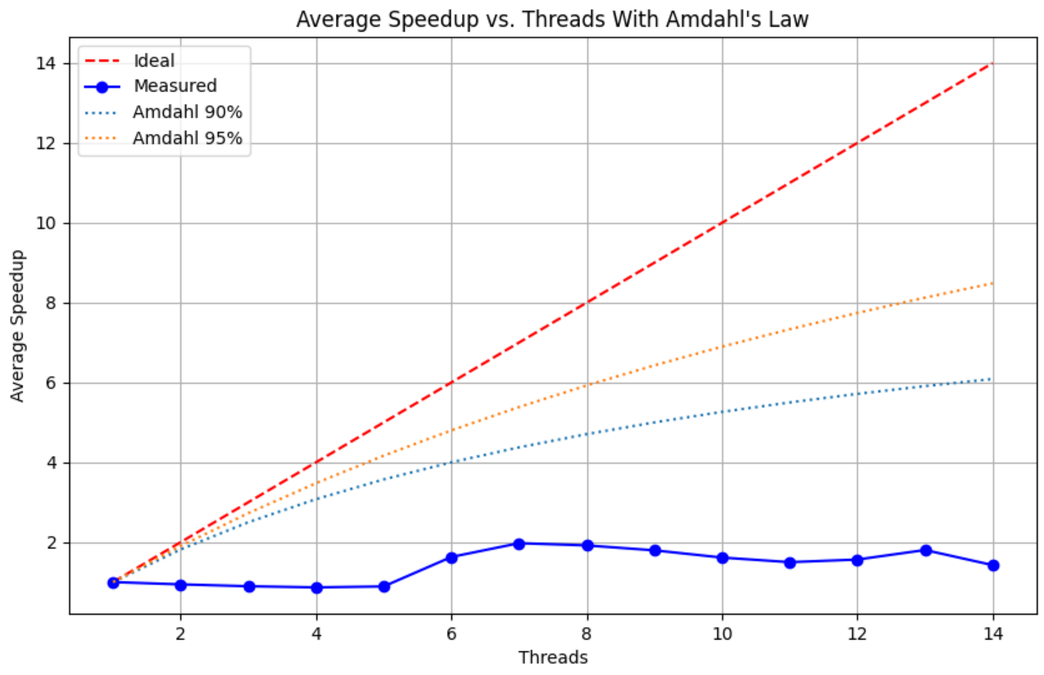
These are threadscope images for cnf_1. We can observe that the optimal depth of 3 led to more efficient spark management, whereas the less optimal depth of 8 resulted in a lot of sparking and fizzling, unnecessary work which subsequently penalized runtime.

Here are speedup results on `UUF100.430.1000/uuf100-01.cnf` with depth-limiting only. DPLL was run with a depth of 8 and threads 1-14 ten times. Speedups were then averaged across each run for each thread count:



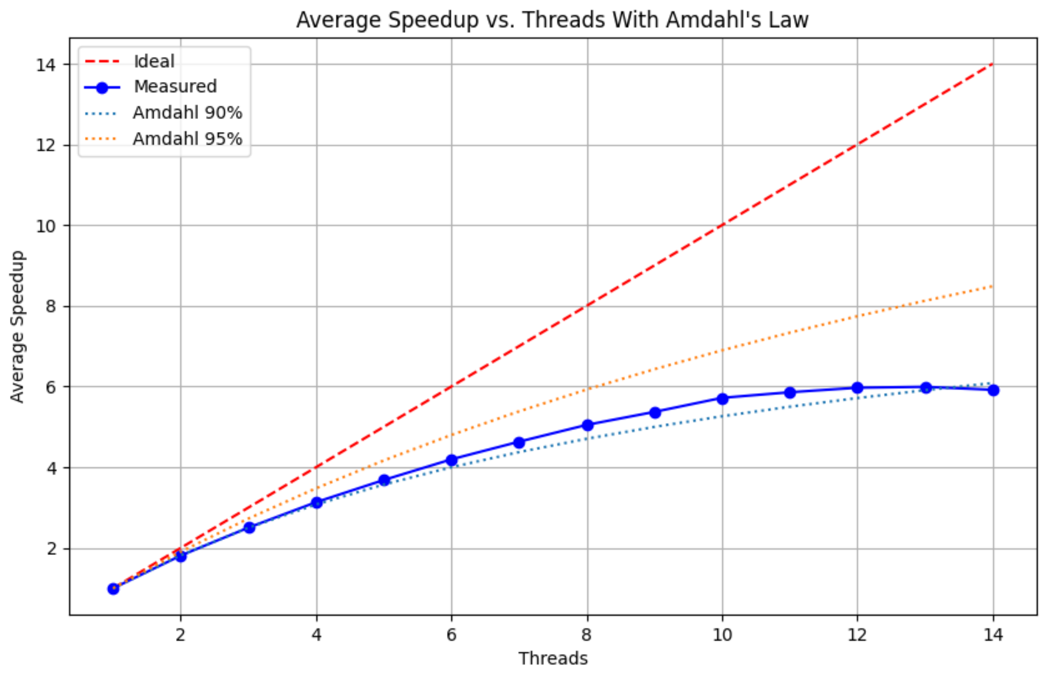
As you can see, our depth-limiting parallel implementation almost achieves theoretical maximum speedup assuming 95% of the work can be parallelized. The tapering at the end of the graph may be due to nonoptimal synchronization, workload balancing, and/or data granularity. Out of the three, workload balancing is the most likely culprit.

Here are speedup results on `UF150.645.100/uf150-01.cnf` with depth-limiting only. DPLL was run with a depth of 25 and threads 1-14 ten times. Speedups were then averaged across each run for each thread count:



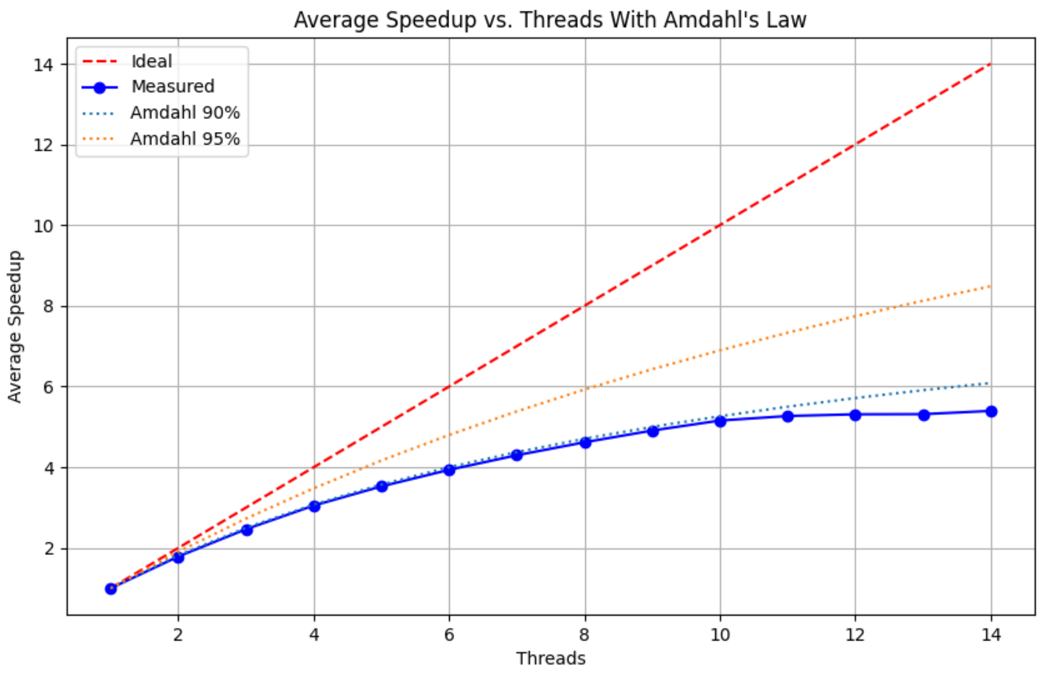
As you can see, our depth-limiting parallel implementation struggles to achieve even near theoretical maximum speedup. This may be due to nonoptimal synchronization, workload balancing, and data granularity. Out of the three, workload balancing and data granularity might be the most culpable. Visualization of this instance's search space may reveal imbalance.

Here are speedup results on `UF150.645.100/uf150-01.cnf` with top-level k-splitting and depth-limiting. DPLL was run with a depth of 1024 and threads 1-14 ten times. Speedups were then averaged across each run for each thread count:



As you can see, our k-splitting implementation is close to achieving theoretical maximum speedup assuming 95% of the work can be parallelized. The tapering may be due to nonoptimal synchronization, workload balancing, and/or data granularity. Out of the three, data granularity may be the most likely culprit, since parameters k and depth were not tuned.

Here are speedup results on `UUF150.645.100/uf150-01.cnf` with top-level k-splitting and depth-limiting. DPLL was run with a depth of 1024 and threads 1-14 ten times. Speedups were then averaged across each run for each thread count:



As you can see, our k-splitting implementation is close to achieving theoretical maximum speedup assuming 95% of the work can be parallelized, although slightly worse compared to `UF150.645.100/uf150-01.cnf`. The tapering may be due to nonoptimal synchronization, workload balancing, and/or data granularity. Out of the three, workload balancing and data granularity may be the most culpable, since parameters k and depth were not tuned and, assuming similar structure, this unsatisfiable instance would require exhaustive search over a similar search space compared to the satisfiable instance, exposing our implementation's deficiencies even more.

5 Conclusion

5.1 Takeaways

Using `Strategies` appears to be the way to go when parallelizing an algorithm in Haskell. Generally, **NP**-complete problems are well suited. In our case, it was able to be applied to a wide range of CNF-SAT problems with decent speedup. However, we found that parallelizing branching is nontrivial. The optimal depth at which parallel branching should be limited must balance under- and over-sparking. Optimizing the underlying sequential implementation of DPLL was also crucial to making improvements in speedup.

5.2 Future Sequential Implementation Work

Implementing branching heuristics is one obvious way we could improve our sequential implementation. Currently, we branch on the first unassigned variable. Other heuristics include greedy algorithms – like selecting the most frequently occurring variable, Bohm's Heuristic, Maximum Occurrences on Minimum sized clauses (MOM), and Jeroslow-Wang's Heuristic – as well as others like Dynamic Largest Combined Sum (DLCS) and Variable State Independent Decaying Sum (VSIDS) which take a more balanced approach. And, as mentioned in Section 2.3, underlying data structures could be optimized as well.

Perhaps even more critical would be the incorporation of conflict driven clause learning. When attempting to backtrack, instead of moving up a single decision level, CDCL-DPLL would perform a defined conflict analysis procedure, (1) deriving a learned clause that prevents the same conflict from occurring in the future and (2) enabling a non-chronological "backjump" to the earliest decision level (not necessarily the most recent) where the conflict could have been avoided. Reducing both the size of the search tree and the time spent backtracking would outweigh additional memory overhead, which could be subsequently mitigated with clause deletion strategies.

5.3 Future Parallel Implementation Work

As mentioned in Section 3.3, further parallel implementation work would involve properly tuning and optimizing k-splitting on a per-level basis. However, other general strategies of this type might be explored. As k-splitting divides the search space into disjoint subspaces to be explored in parallel, more advanced heuristics which determine the split could be implemented, like scattering functions, XOR constraints, or most notably, guiding paths [5]. Guiding paths would require the addition of Boolean flags, indicating whether both true/false assignments of a given variable have been tried, to the existing data structure that tracks assignments.

The critical weakness of guiding paths is that they cannot predict the hardness of the generated subspaces, which then requires dynamic load balancing (as already suggested in Section 4). Commonly, this would be done via dynamic work stealing. As found via our third parallel implementation and by the weakness of guiding paths, the effectiveness of search space splitting relies on effective selection and application of splitting heuristics, as well as load balancing. But what about non-randomly generated input instances? Input instances from the real world often have huge numbers of variables with often highly imbalanced search spaces. Formula decomposition, portfolio, or hierarchical collaboration external parallelization strategies might be tried next. Even more ambitious would be to employ an internal parallelization strategy, taking a manager-worker or task-queues approach to distribute work among threads, or to employ a hybrid external-internal strategy [6].

6 Code Listing

6.1 app/Main.hs

```haskell
module Main (main) where

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

import DIMACSParser
import Lib(constructTree, satisfyParDPLL)
import qualified Data.Vector as V

main :: IO ()
main = do
  args <- getArgs
  case args of
    [filename] -> do
      content <- readFile filename
      case parseDIMACS content of
        Left err -> die $ "Error parsing DIMACS input file: " ++ err
        Right (numVars, clauses) -> do
          let cnf = constructTree clauses
          let initialAssignment = V.replicate numVars Nothing
              (dpllResult, assignment) = satisfyParDPLL 40 cnf initialAssignment
          putStrLn $ "DPLL Solver result: "
              ++ if dpllResult then "SAT: " ++ (show assignment) else "UNSAT"
    _ -> do
      progName <- getProgName
      die $ "Usage: " ++ progName ++ " <filename>"
```

6.2 src/DIMACSParser.hs

```haskell
module DIMACSParser (parseDIMACS) where

import Lib(Clauses(..), Literal(..))
import qualified Data.List as L
import Data.Char (isDigit)

-- Parse given DIMACS input file into `(numVars, [Clause])`
parseDIMACS :: String -> Either String (Int, [Clauses])
parseDIMACS content = do
    let (_, remainingLines) = span (L.isPrefixOf "c") (lines content)
    case remainingLines of
         (problemLine:clauseLines) -> do
           (numVars, numClauses) <- parseProblemLine problemLine
           clauses <- parseClauses numVars numClauses clauseLines
           return (numVars, clauses)
         _ -> Left "Error: missing problem line and clauses"

-- Parse expected problem line
parseProblemLine :: String -> Either String (Int, Int)
parseProblemLine line =
  case words line of
    ["p", "cnf", vars, clauses] | all (all isDigit) [vars, clauses] ->
      Right (read vars, read clauses)
    _ -> Left "Error: invalid or missing problem line"

-- Parse and validate clause lines
parseClauses :: Int -> Int -> [String] -> Either String [Clauses]
parseClauses numVars numClauses clauseLines
  | length clauseLines /= numClauses = Left $ "Error: expected "
                                        ++ show numClauses
                                        ++ " clauses, found "
                                        ++ show (length clauseLines)
  | otherwise = mapM (parseClause numVars) clauseLines

-- Parse a single clause
parseClause :: Int -> String -> Either String Clauses
parseClause numVars line = do
  let literalStrings = init $ words line -- Drop trailing '0'
  literals <- mapM parseLiteral literalStrings
  return $ Clause literals
  where
    -- Parse a single literal within a clause
    parseLiteral :: String -> Either String Literal
    parseLiteral lString
  | head lString == '-' && (all isDigit) (tail lString) =
      validateBounds ((read (tail lString) :: Int)) LiteralNeg
  | (all isDigit) lString =
      validateBounds ((read lString :: Int)) LiteralPos
  | otherwise =
      Left $ "Error: invalid literal " ++ lString

-- Ensure a clause doesn't contain more variables than declared in problem line
validateBounds :: Int -> (Int -> Literal) -> Either String Literal
validateBounds lInt constructor
  | lInt < 1 || lInt > numVars = Left $ "Error: invalid literal "
                                   ++ show lInt ++ " for "
                                   ++ show numVars ++ " variables"
  | otherwise = Right $ constructor (lInt-1)
```

6.3 src/Lib.hs

```haskell
module Lib
    ( ExpressionTree(..)
    , Clauses(..)
    , Literal(..)
    , parDpll
    , satisfyParDPLL
    , constructTree
    ) where

import qualified Data.Vector as V
import Control.Parallel.Strategies(Strategy, using, rpar)
import Control.DeepSeq(NFData)

---------------------------------
-- Data Types: CNF Representation
---------------------------------
data ExpressionTree = Expr [Clauses]
  deriving (Show, Read, Eq)

data Clauses = Clause [Literal]
  deriving (Show, Read, Eq)

data Literal = LiteralPos Int | LiteralNeg Int
  deriving (Show, Read, Eq)

-- Helper function: construct CNF tree representation from clauses
constructTree :: [Clauses] -> ExpressionTree
constructTree clauses = Expr clauses

-------------------
-- Helper functions
-------------------
-- Check if formula is empty (contains no clauses) i.e. SAT with current assignment
isFormulaEmpty :: ExpressionTree -> Bool
isFormulaEmpty (Expr clauses) = null clauses

-- Check if formula contains an empty clauses i.e. UNSAT with current assignment
containsEmptyClause :: ExpressionTree -> Bool
containsEmptyClause (Expr clauses) = any (\(Clause lits) -> null lits) clauses

-- Apply a single variable assignment to the formula
applyAssignment :: ExpressionTree -> (Int, Bool) -> ExpressionTree
applyAssignment (Expr clauses) (var, val) =
  Expr $ map (removeContradictions var val) $ filter (not . satisfiedBy var val) clauses
  where
    satisfiedBy v b (Clause lits) = any (literalMatches v b) lits
    literalMatches v b (LiteralPos x) = x == v && b == True
    literalMatches v b (LiteralNeg x) = x == v && b == False

    removeContradictions v b (Clause lits) =
      Clause (filter (\l -> not $ literalContradicts v b l) lits)
    literalContradicts v b (LiteralPos x) = x == v && b == False
    literalContradicts v b (LiteralNeg x) = x == v && b == True

-- Find a unit clause
findUnitClause :: ExpressionTree -> Maybe Literal
findUnitClause (Expr clauses) =
  case [lits | Clause lits <- clauses, length lits == 1] of
    ((lit:_):_) -> Just lit
    _           -> Nothing

-- Find a pure literal
findPureLiteral :: ExpressionTree -> Maybe (Int, Bool)
findPureLiteral (Expr clauses) =
  let allLits = concatMap (\(Clause ls) -> ls) clauses
      (posVars, negVars) = foldr countPolarity ([],[]) allLits
      countPolarity (LiteralPos v) (ps,ns) = (v:ps, ns)
      countPolarity (LiteralNeg v) (ps,ns) = (ps, v:ns)
      uniquePos = filter (`notElem` negVars) posVars
      uniqueNeg = filter (`notElem` posVars) negVars
  in case uniquePos of
      (v:_) -> Just (v,True)
      []    -> case uniqueNeg of
                 (v:_) -> Just (v,False)
                 []    -> Nothing

-- Set a variable in the assignment vector
setAssignment :: V.Vector (Maybe Bool) -> Int -> Bool -> V.Vector (Maybe Bool)
setAssignment asg var val = asg V.// [(var, Just val)]

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

satisfyParDPLL :: Int
                  -> ExpressionTree
```

```haskell
                    -> V.Vector (Maybe Bool)
                    -> (Bool, V.Vector (Maybe Bool))
satisfyParDPLL = parDpll pairParStrat

specialOr :: [(Bool,V.Vector (Maybe Bool))] -> (Bool, V.Vector (Maybe Bool))
specialOr ((b,vec):bs) = if b then (True, vec) else specialOr bs
specialOr [] = (False,V.empty)

-----------------------
-- Parallel DPLL Solver
-----------------------
parDpll :: Strategy[(Bool, V.Vector (Maybe Bool))]
             -> Int
             -> ExpressionTree
             -> V.Vector (Maybe Bool)
             -> (Bool, V.Vector (Maybe Bool))
parDpll _ 0 formula assignment = dpll formula assignment
parDpll strat d formula assignment
  | isFormulaEmpty formula = (True, assignment)     -- Base case: SAT
  | containsEmptyClause formula = (False, V.empty)  -- Base case: UNSAT
  | Just unit <- findUnitClause formula =           -- Unit Propagation
      let (var, val) = case unit of
                         LiteralPos v -> (v, True)
                         LiteralNeg v -> (v, False)
          newAsg =
            setAssignment assignment var val
          newFormula =
            applyAssignment formula (var, val) in parDpll strat d newFormula newAsg
  | Just (v,b) <- findPureLiteral formula =         -- Pure Literal Elimination
      let newAsg = setAssignment assignment v b
          newFormula = applyAssignment formula (v, b) in parDpll strat d newFormula newAsg
  | otherwise = case formula of                     -- Branch
        Expr (Clause c:_) ->
          let pickLiteral = head c                  -- Branching Heuristic: naive
              varToAssign = case pickLiteral of
                              LiteralPos v -> v
                              LiteralNeg v -> v
              tryTrueAsg  = setAssignment assignment varToAssign True
              tryTrueForm = applyAssignment formula (varToAssign, True)
              tryFalseAsg = setAssignment assignment varToAssign False
              tryFalseForm = applyAssignment formula (varToAssign, False)
              satFalse = parDpll strat (d-1) tryFalseForm tryFalseAsg
              satTrue  = parDpll strat (d-1) tryTrueForm tryTrueAsg
              in specialOr ([satTrue, satFalse] `using` strat)
        Expr [] -> (False, V.empty)
```

```haskell
-------------------------
-- Sequential DPLL Solver
-------------------------
dpll :: ExpressionTree -> V.Vector (Maybe Bool) -> (Bool, V.Vector (Maybe Bool))
dpll formula assignment
  | isFormulaEmpty formula = (True, assignment)      -- Base case: SAT
  | containsEmptyClause formula = (False, V.empty)  -- Base case: UNSAT
  | Just unit <- findUnitClause formula =            -- Unit Propagation
      let (var, val) = case unit of
                         LiteralPos v -> (v, True)
                         LiteralNeg v -> (v, False)
          newAsg = setAssignment assignment var val
          newFormula = applyAssignment formula (var, val) in dpll newFormula newAsg
  | Just (v,b) <- findPureLiteral formula =          -- Pure Literal Elimination
      let newAsg = setAssignment assignment v b
          newFormula = applyAssignment formula (v, b) in dpll newFormula newAsg
  | otherwise = case formula of                      -- Branch
        Expr (Clause c:_) ->
          let pickLiteral = head c                   -- Branching Heuristic: naive
              varToAssign = case pickLiteral of
                              LiteralPos v -> v
                              LiteralNeg v -> v
              tryTrueAsg  = setAssignment assignment varToAssign True
              tryTrueForm = applyAssignment formula (varToAssign, True)
              (satTrue, trueAsgt) = dpll tryTrueForm tryTrueAsg
              in if satTrue
                 then (True, trueAsgt)
                 else
                  let tryFalseAsg  =
                                        setAssignment assignment varToAssign False
                      tryFalseForm =
                                        applyAssignment formula (varToAssign, False)
                      (satFalse, falseAsgt) =
                                        dpll tryFalseForm tryFalseAsg in
                                           if not satFalse then (False, V.empty)
                                           else (True, falseAsgt)
        Expr [] -> (False, V.empty)
```

References

[1] `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

[2] `https://en.wikipedia.org/wiki/DPLL_algorithm`

[3] `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

[4] `https://cca.informatik.uni-freiburg.de/papers/BiereFallerFazekasFleuryFroleyksPollitt-SAT-Competition-2024-solvers.pdf`

[5] G. Chu, P.J. Stuckey, and A. Harwood. Pminisat - A parallelization of MiniSat 2.0. *SAT race*, 2008.

[6] S Li. Boolean Satisfiability: Parallelization and Exploration. *UDSpace*, 2015.