Parallel SAT Solver

COMS 4995 Parallel Functional Programming
Yixuan Li(yl3803), Jiaqian Li(cl4283), Phoebe Wang(kw3036)

1 Abstract

The **Boolean Satisfiability Problem (SAT)** is an NP-complete problem widely used in areas such as formal verification, artificial intelligence, and cryptography. SAT is an excellent candidate for parallelization because its search space can be naturally divided into independent subproblems. When branching on a variable, each branch can be explored concurrently, as the assignments for different branches do not interfere with each other.

In our final project, we explore multiple approaches to solving SAT, starting with a parallel brute-force method and then implementing the **DPLL** (Davis-Putnam-Logemann-Loveland) algorithm, a more efficient SAT-solving approach. We parallelize DPLL using Haskell's `parMap` and `parListChunk` strategies to distribute subproblems across multiple threads, as well as a worker queue strategy, which splits tasks and adds them to a shared task queue for parallel processing. Both brute-force and DPLL parallel implementations demonstrate performance improvements compared to running the program with a single thread, proving the power of parallel processing.

2 Methods

2.1 SAT Problem Definition

The Boolean Satisfiability Problem (SAT) asks whether there exists an assignment of true or false values to variables that satisfies a given Boolean formula, typically expressed in Conjunctive Normal Form (CNF): A CNF formula is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals.

Example: $(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3)$.

The primary goal of our project is to improve the traditional approaches for solving the SAT problem using parallelization. By dividing the search space into independent subproblems and solving each of them in parallel, we hope to achieve performance improvements over sequential methods.

2.2 Brute Force Method

2.2.1 Description

The naive approach is to generate all 2^n possible truth assignments for n variables. Each assignment is evaluated against the CNF formula to determine if it satisfies all clauses. To parallelize this brute-force method, we divide the 2^n possible assignments into k chunks, where k corresponds to the number of threads or available cores. Next, each chunk of assignments is evaluated concurrently using Haskell's `parMap` or `parListChunk` strategies, and the results from all threads are combined to return the first satisfying solution.

`parListChunk` is different from the `parList` strategy we discussed in class. `parListChunk` is particularly effective for this task as it processes chunks of assignments in parallel while evaluating each chunk sequentially. This approach improves load balancing and reduces overhead.

The brute-force method evaluates all 2^n possible assignments for n variables to determine if any satisfies the given Boolean formula. For each assignment, the algorithm checks the formula by iterating over its clauses and literals. If there are m clauses and each clause contains at most k literals, the complexity for the sequential brute force is approximately $O(2^n * m * k)$.

When the brute-force method is parallelized, the search space of 2^n assignments is divided into p chunks. Each thread evaluates $\frac{2^n}{p}$ assignments independently. The parallelized time complexity for each thread is reduced to $O(\frac{2^n}{p} * m * k)$. Overall, this brute-force approach is simple but inefficient for large problems due to exponential growth of the search space with n.

2.2.2 Implementation Details

1. Generate all possible truth assignments for n variables and split them into smaller chunks for parallel processing.

```haskell
generateAllAssignments :: Int -> [[Assignment]]
generateAllAssignments n =
  let allAssignments = [ zip [1..n] bools | bools <- replicateM n [False, True] ]
      chunkSize = 128
  in chunksOf chunkSize allAssignments
```

2. Check whether a literal is satisfied under the current assignment.

```haskell
evaluateLiteral :: Assignment -> Literal -> Bool
evaluateLiteral assignment lit =
  let variable = abs lit
      value = fromJust (lookup variable assignment)
  in if lit > 0 then value else not value
```

For assignment $= [(1, False), (2, False)]$ and $lit = -1$, the result is $True$.

3. Check whether a clause is satisfied under the current assignment.

```haskell
evaluateClause :: Assignment -> Clause -> Bool
evaluateClause assignment clause = any (evaluateLiteral assignment) clause
```

For assignment $= [(1, False), (2, False)]$ and clause = [-1, 2], the result is $True$ because 2 is $False$, but -1 is $True$.

4. Check whether the CNF formula is satisfied under the current assignment.

```haskell
evaluateCNF :: Assignment -> CNF -> Bool
evaluateCNF assignment cnf = all (evaluateClause assignment) cnf
```

5. Check a chunk of truth assignments to find a satisfying assignment for the given CNF formula.

```haskell
evaluateChunk :: CNF -> [Assignment] -> Maybe Assignment
evaluateChunk cnf assignments =
    findFirstSatisfying assignments
  where
    findFirstSatisfying [] = Nothing
    findFirstSatisfying (assign:rest)
        | evaluateCNF assign cnf = Just assign
        | otherwise = findFirstSatisfying rest
```

6. Solve the SAT problem in parallel with `parMap` and use the < | > operator to terminate the search once a satisfying assignment is found.

```
solveSATParallel :: CNF -> Int -> Maybe Assignment
solveSATParallel cnf numVars =
  let chunks = generateAllAssignments(numVars)
      results = parMap rdeepseq (evaluateChunk cnf) chunks
  in foldr (<|>) Nothing results
```

2.3 DPLL Algorithm

Description

The DPLL algorithm is a backtracking-based algorithm for the SAT problem. The algorithm starts by *iteratively* applying **unit propagation**, which assigns truth values to variables in unit clauses (clauses with only one unassigned literal) to satisfy those clauses. After simplifying the formula, the algorithm proceeds by choosing a variable (based on some heuristic such as the Variable State Independent Decaying Sum), making a **binary decision** (assigning true or false), and *recursively* exploring the resulting subproblems. If a conflict is encountered, the algorithm **backtracks** to the previous decision level and explores the alternative branch. If all branches are exhausted without finding a solution, the formula is declared unsatisfiable. Otherwise, a satisfying assignment is returned.

Compared with the brute-force method implemented in Section 2.2, the DPLL algorithm is more space-efficient. It explores the search tree in a systematic, depth-first manner rather than explicitly generating all possible assignments upfront. The unit-propagation together with the variable selection heuristic helps the SAT solver focus on more promising parts of the search space and thus skip redundant computation. This allows more advanced parallel strategies such as dynamic work-stealing and search space partitioning.

Implementation Details

1. The main data structure used in the SAT solver is defined as follows:

```
data SatSolver = SatSolver
  { clauses :: ![Clause],              -- Clauses to solve
    bindings :: !(IM.IntMap Bool)     -- Current variable assignments
  }
```

Fields:

- `clauses`: List of clauses in CNF.
- `bindings`: Mapping of variable indices to their truth values.

2. The `solve` function first simplifies the solver then solves recursively until a solution is found or proven unsatisfiable:

```
solve :: (Monad m, Alternative m) => SatSolver -> m SatSolver
solve solver = maybe empty solveRecursively (simplify solver)
```

3. The formula is solved recursively by branching on variables and exploring both truth assignments:

```
solveRecursively :: (Monad m, Alternative m) => SatSolver -> m SatSolver
solveRecursively solver
    | isSolved solver = pure solver
```

```haskell
    | otherwise = do
        let varToBranch = selectBranchVar solver
        branchOnUnbound varToBranch solver >>= solveRecursively
```

For the variable selection, we choose the first literal from the shortest clause (which can be changed to other heuristic):

```haskell
selectBranchVar :: SatSolver -> Var
selectBranchVar solver =
    var $ head $ literals $ head $ sortBy shorterClause (clauses solver)
```

Both `True` and `False` assignments for a variable are explored:

```haskell
branchOnUnbound :: (Monad m, Alternative m) => Var -> SatSolver -> m SatSolver
branchOnUnbound name solver =
    guessAndRecurse (mkLit name True) solver
    <|>
    guessAndRecurse (mkLit name False) solver
```

4. Upon guessing the value of a literal, we can simplify the formula by iteratively using unit propagation:

```haskell
simplify :: (Monad m, Alternative m) => SatSolver -> m SatSolver
simplify solver = do
    case findUnitClause (clauses solver) of
        Nothing -> pure solver
        Just lit -> do
            let updatedSolver = solver {
            bindings = IM.insert (var lit) (not (sign lit)) (bindings solver) }
            case propagate lit (clauses updatedSolver) of
                Nothing -> empty
                Just updatedClauses -> simplify $ updatedSolver { clauses = updatedClauses }

propagate :: Lit -> [Clause] -> Maybe [Clause]
propagate lit inputClauses =
    let updatedClauses = mapMaybe (processClause lit) inputClauses
    in if any (null . literals) updatedClauses
       then Nothing
       else Just updatedClauses
```

2.4 Parallel DPLL

2.4.1 Static Parallelism

Description

In our parallel implementation of the DPLL algorithm, the search space is divided by branching on a small subset of variables at the start. Each combination of truth assignments for these variables defines an independent subproblem, which is assigned to a separate thread for evaluation. To distribute the subproblems , we used Haskell's `parMap` and `parListChunk` strategies. While `parMap` evaluates all subproblems concurrently, `parListChunk` processes larger batches of subproblems sequentially within each thread while evaluating the chunks in parallel across threads. Once the subproblems are distributed, each thread executes the DPLL algorithm independently on its assigned subproblem. The solver terminates as soon as the first satisfying assignment is found or concludes that the formula is unsatisfiable after exploring all subproblems.

The time complexity of the parallel DPLL algorithm remains exponential in the worst case, as the SAT problem is NP-complete. If the search space is divided into k independent subproblems, the theoretical time complexity per thread is reduced to: $O(\frac{2^n}{k})$ where n is the number of variables, and k is the number of threads or subproblems. This assumes perfect load balancing and no overhead.

Implementation Details

1. Randomly select 5 variables based on index from the SAT problem to use for branching.

```
selectRandomVars :: StdGen -> SatSolver -> [Var]
selectRandomVars gen solver =
    let allVars = IS.toList $ IS.fromList
                  [var lit | clause <- clauses solver, lit <- literals clause]
        indices = take 5 $ randomRs (0, length allVars - 1) gen -- take 5 random vars
    in map (allVars !!) indices
```

2. Generate a list of subproblems by applying all possible truth assignments to the selected variables. We use `mapMaybe` to apply `applyAssignment` to each assignment in the list, and discard the assignment if `applyAssignment` returns Nothing.

```
generateSubproblems :: [Var] -> SatSolver -> [SatSolver]
generateSubproblems vars solver =
    -- some assignments may fail due to conflicts, filter them
    mapMaybe (`applyAssignment` solver) (generateAssignments vars)
```

3. Generate all possible truth assignments for a given list of variables. We use sequence to produce the Cartesian product of truth values to get 2^k assignments for k variables.

```
generateAssignments :: [Var] -> [[Lit]]
generateAssignments vars =
    [ [mkLit v val | (v, val) <- zip vars vals] |
            vals <- sequence (replicate (length vars) [True, False]) ]
```

If $lengthvars = 3$, replicate will produce $[[T, F], [T, F], [T, F]]$, and sequence will produce all 8 assignments. E.g. $[[T, T, T], [T, T, F], \dots]$

4. Apply the assignment, which is a list of literals, to the SAT solver, resulting in a new solver state or Nothing if there is conflict. We use `foldM` to iterate over the list of literals (lits), applying each one to the SAT solver (baseSolver) using the function guess from sequential DPLL.

```
applyAssignment :: [Lit] -> SatSolver -> Maybe SatSolver
applyAssignment lits baseSolver =
    foldM (\solver lit -> guess lit solver) baseSolver lits
```

5. Solve the SAT problem in parallel with `parMap`. We use `mapMaybe` to collect the results that are `Just` values and `listToMaybe` to return the first element if the list is not empty.

```
parallelSolveOne :: StdGen -> SatSolver -> Maybe SatSolver
parallelSolveOne gen solver =
    let vars = selectRandomVars gen solver
        subproblems = generateSubproblems vars solver
        results = parMap rdeepseq solve subproblems
    in listToMaybe (mapMaybe id results) -- return first solution
```

2.4.2 Worker Queue Strategy

Description

For the worker queue strategy, the search space is divided by branching on a small subset of variables at the start. Instead of directly assigning these subproblems to threads, they are added to a shared task queue. Multiple threads are launched to fetch subproblems from this queue and execute the DPLL algorithm independently. This ensures a dynamic workload distribution, as threads will fetch new tasks as soon as they complete their current ones. The shared queue is managed using Haskell's `Software Transactional Memory (STM)` primitives: `writeTQueue` is used to add tasks, and `tryReadTQueue` allows threads to fetch tasks atomically, both wrapped in `atomically` blocks to ensure thread-safe operations.

Unlike static strategies like `parMap` or `parListChunk`, where all subproblems are distributed upfront, the worker queue allows idle threads to pick up remaining tasks dynamically, avoiding load imbalance. When one thread finds a satisfying assignment, it updates a shared result variable to terminate all threads immediately, reducing redundant computation. If no solution is found, the algorithm guarantees all subproblems are explored before concluding unsatisfiability.

The worker queue's time complexity is also $O(2^n)$ in the worst case, but its dynamic nature significantly improves practical runtime by balancing computation across threads. However, the worker queue dynamically adapts to the workload, making the practical runtime more efficient compared to static approaches. Although the task synchronization adds a small overhead, the dynamic load balancing makes this strategy effective for solving large SAT problems, especially when the workload is uneven.

Implementation Details

1. Select random variables by `selectRandomVars` for branching and use `generateSubproblems` to find all possible truth assignments for the selected variables.
(For details about `selectRandomVars`, `generateSubproblems` and related functions, see Implementation Details 1-4 in Static Parallelism)

```
let vars = selectRandomVars gen solver
let subproblems = generateSubproblems vars solver
```

2. Initialize the shared task queue `TQueue` to store the generated subproblems. The subproblems are added to the task queue `atomically`, and multiple worker threads are launched using `forkIO` to process the tasks concurrently.

```
taskQueue <- newTQueueIO
atomically $ mapM_ (writeTQueue taskQueue) subproblems
replicateM_ numThreads $ forkIO $ worker taskQueue resultsVar
```

3. Each thread repeatedly fetches tasks from the shared queue and attempts to solve them using the solve function. If a solution is found, it is store in the results variable `resultsVar` to signal termination. If the task fails, the thread fetches the next task from the queue.

```
takeMVar resultsVar -- blocked until a result is added
worker taskQueue resultsVar = do
    maybeTask <- atomically $ tryReadTQueue taskQueue -- read a task
    case maybeTask of
        Nothing -> return () -- exit with no left work
        Just subproblem -> do
            let result = solve subproblem
            case result of
```

```haskell
                    Just solution -> putMVar resultsVar (Just solution)
                    Nothing -> worker taskQueue resultsVar -- keep working
```

The solver terminates as soon as a solution is found, or all tasks in the queue are explored with no satisfying solutions.

2.5 SAT Solver Test Generator

Description

In `SatGen.hs`, we implemented a generator that creates a satisfiable Conjunctive Normal Form (CNF) formula to test the SAT solver. The generator takes three inputs: the number of variables (`numVars`), the number of clauses (`numClauses`), and the number of literals per clause (`clauseLen`). For each clause, a satisfying literal is chosen based on this assignment, and the rest of the literals are randomly generated to meet the specified clause length. Duplicate clauses are removed to ensure the output is clean. The resulting CNF formula is guaranteed to be satisfiable and can be output in DIMACS format, which is a standard format for SAT solvers.

Implementation Details

1. In the generator, we have three data types:

- `Literal`: represents a positive or negative variable.
- `Clause`: A list of literals forming a single clause.
- `CNF`: A list of clauses that form the final formula.

2. Generates a random literal with a variable index and sign by `randomLiteral`.

```haskell
randomLiteral :: Int -> IO Literal
randomLiteral numVars = do
  var <- randomRIO (1, numVars)
  sign <- randomRIO (False, True)
  return $ if sign then var else -var
```

3. Generates a clause that includes a specific satisfying literal `satLit` by `generateSatisfiableCNF`. This function ensures no duplicate or negated literals appear within the same clause.

```haskell
generateClauseWithSat :: Int -> Int -> Literal -> IO Clause
generateClauseWithSat numVars len satLit = go (Set.singleton satLit)
  where
    go used
      | Set.size used == len = pure $ Set.toList used
      | otherwise = do
          lit <- randomLiteral numVars
          if Set.member lit used || Set.member (-lit) used
            then go used
            else go (Set.insert lit used)
```

4. Creates a list of clauses that together form a satisfiable CNF in `generateSatisfiableCNF`. It generates a random truth assignment for all variables, and then constructs each clause using the `generateClauseWithSat` function.

```haskell
generateSatisfiableCNF :: Int -> Int -> Int -> IO CNF
generateSatisfiableCNF numVars numClauses clauseLen =
  do
    randVals <- replicateM numVars (randomRIO (0, 1) :: IO Int)
```

```haskell
  -- convert from numbers to booleans
  let assignment = map (==1) randVals
  let satisfying = zipWith (\v b -> if b then v else -v) [1..numVars] assignment
  clauses <- replicateM numClauses $ do
    -- pick a random satisfying literal
    satLit <- (satisfying !!) <$> randomRIO (0, numVars - 1)
    -- genrate the clause using this literal
    generateClauseWithSat numVars clauseLen satLit
  -- remove duplicates
  pure $ Set.toList $ Set.fromList clauses
```

A random truth assignment is generated at the start to determine the satisfying literals for the clauses; each clause includes one satisfying literal to ensure it evaluates to True, along with other random literals. Duplicate clauses are removed using a set to produce a clean CNF formula.
5. Converts the CNF formula into the standard DIMACS format for SAT solvers by `cnfToDimacs`.

```haskell
cnfToDimacs :: Int -> CNF -> String
cnfToDimacs numVars cnf =
  let
    header = "p cnf " ++ show numVars ++ " " ++ show (length cnf)
    clauseToString :: Clause -> String
    clauseToString clause =
        let numbers = map show clause -- convert numbers to strings
            joined = unwords numbers -- join with spaces
        in joined ++ " 0"
    clauseStrings = map clauseToString cnf
    allLines = header : clauseStrings
    in
      unlines allLines
```

3 Evaluation

3.1 Environment Setup

The experiments were conducted on a machine equipped with an **Apple M3 Pro processor** featuring **11 physical cores and threads**, supported by **18 GB of RAM**. This provides us enough computational power for parallel SAT solving.

For the benchmark problems, we developed two sets of test data using our custom SAT generator. Due to the high memory and time complexity of the brute-force approach, it was evaluated on a smaller SAT problem consisting of 25 variables, 75 clauses, and 5 literals per clause. In contrast, the more efficient DPLL algorithm, designed to handle larger and more complex inputs, was tested on significantly larger problems with 100 variables, 50,000 clauses, and 5 literals per clause. This distinction in problem sizes allowed us to demonstrate the scalability of the DPLL solver compared to the brute-force approach and to evaluate the effectiveness of parallelism in both methods.

3.2 Results

3.2.1 Brute Force Method

The basic brute-force parallel solver was evaluated using different parallel strategies `parMap` and `parListChunk` across varying numbers of threads and chunk sizes. The CNF formula used for testing consisted of **25 variables**, **75 clauses**, and **5 literals per clause**.

When running the solver with `parMap` and 128 chunks across 1 to 11 threads, the results showed significant fluctuations in speedup. The peak speedup achieved was 1.2x at 4 threads, but the performance dropped inconsistently as the number of threads increased. The following Speedup Graph shows that the performance does not consistently improve as more threads are added, and a potential reason may be the garbage collection (GC) takes up a significant amount of time, slowing down the solver.

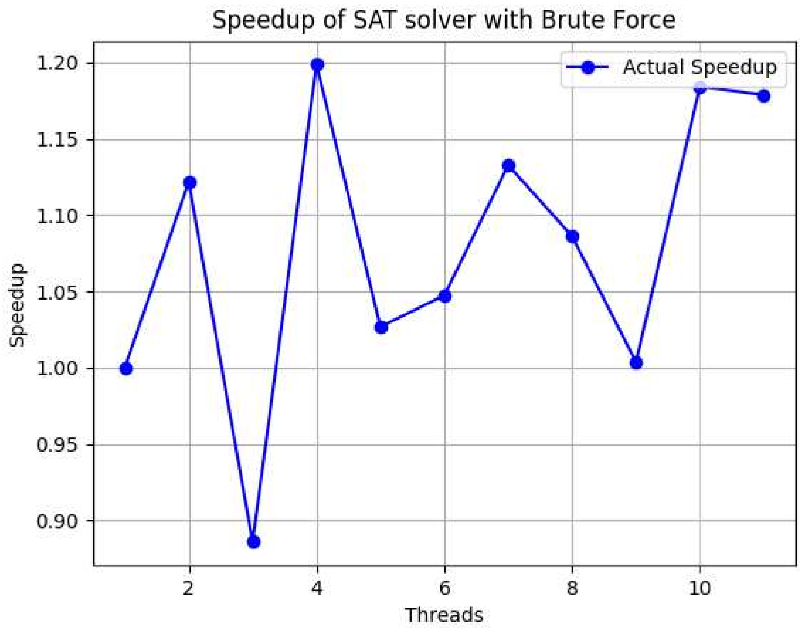


Figure 1: Brute-Force Speedup

To analyze the impact of chunk sizes, the solver was executed with `parListChunk` using chunk sizes of 16, 32, and 64 on 8 threads. The results showed a clear trade-off between load balancing and GC performance. With **smaller chunk sizes**, such as **16**, the **total runtime** was **6.885s**, but the **garbage collection time** accounted for **5.953s**, indicating that frequent memory management was a significant bottleneck.

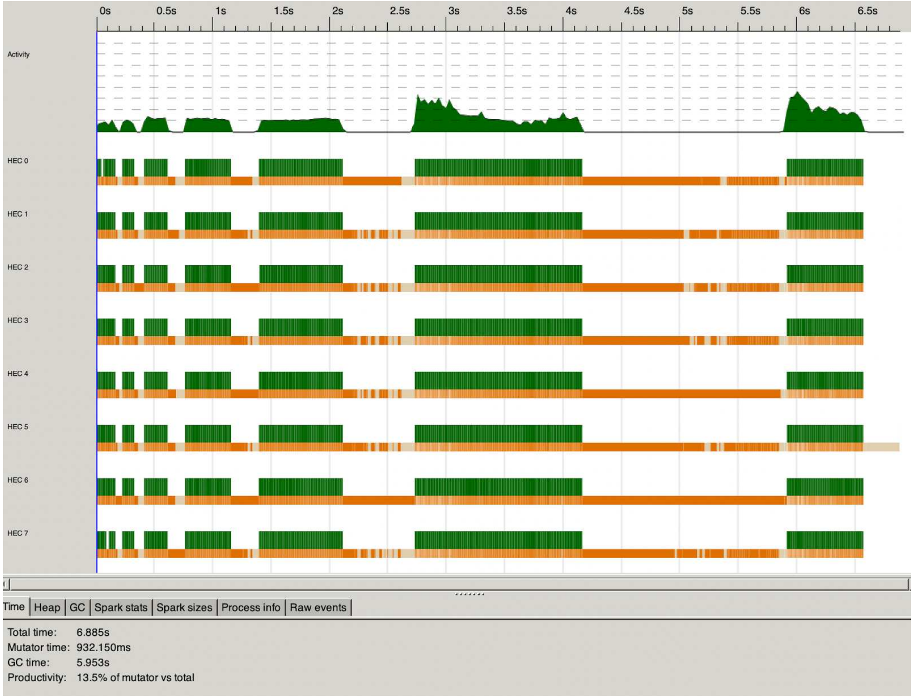


Figure 2: `parListChunk 4 rdeepseq`: chunk size = 16

Increasing the chunk size to **32** resulted in the best overall performance, with a **total runtime of 6.029s** and a reduced **GC time of 5.048s**.

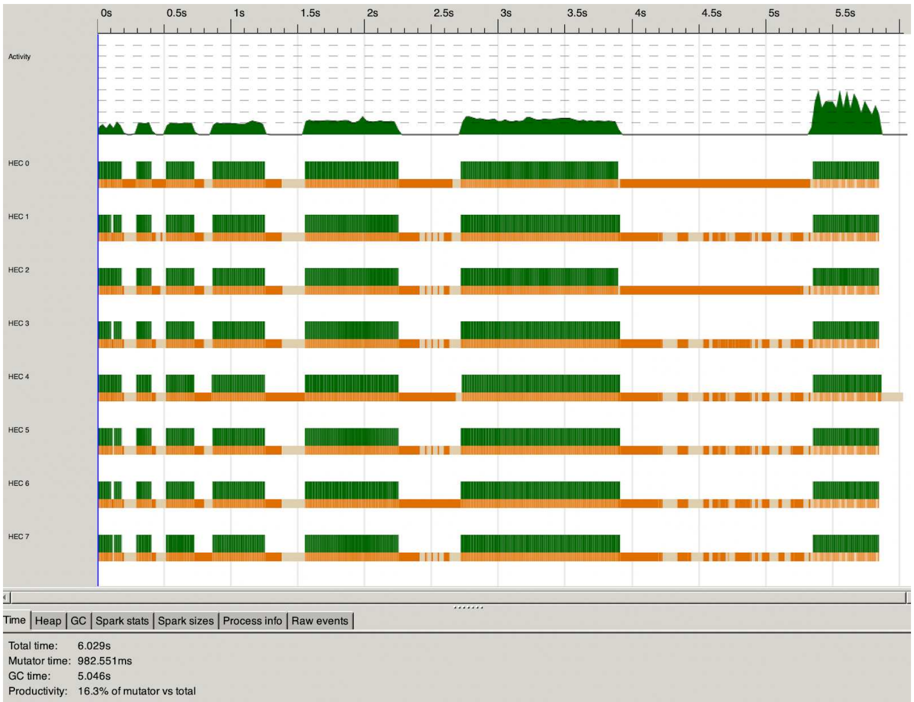


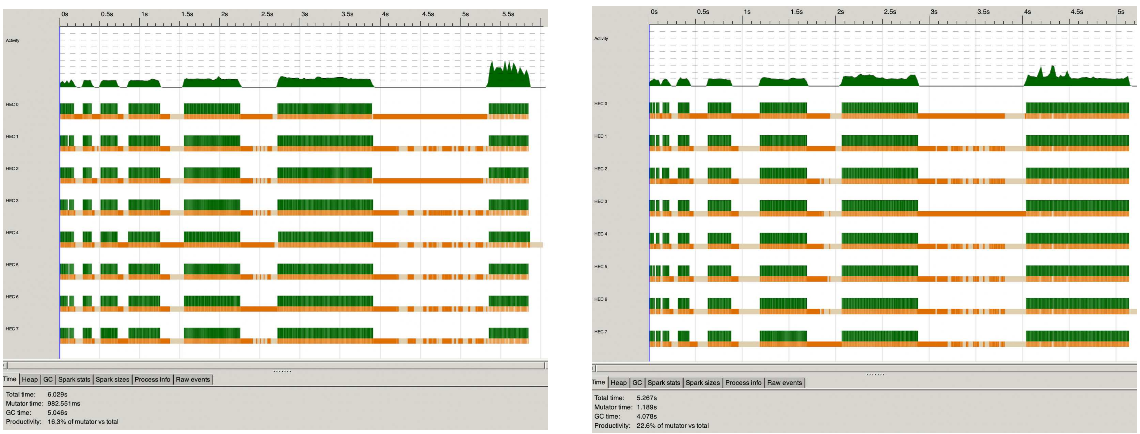
Figure 3: `parListChunk 4 rdeepseq`: chunk size = 32

However, when the chunk size was increased to **64**, the **GC time dropped significantly to 2.971s**, but the **overall runtime increased to 7.525s** due to uneven load distribution, as threads became underutilized.

Figure 4: `parListChunk 4 rdeepseq`: chunk size = 64

This demonstrates that while larger chunks reduce GC overhead, they can lead to poor parallel efficiency if the workload is not evenly distributed.

And now, when comparing the two strategies, `parMap` outperformed `parListChunk` in terms of runtime. With the same input and chunk configuration, `parMap` achieved a **runtime of 5.267s** and **GC time of 4.078s**, while `parListChunk` with chunk size 128 required **runtime of 6.029s** and **GC time of 5.048s** with higher GC overhead. The better performance of `parMap` can be attributed to its ability to evaluate all subproblems concurrently, whereas `parListChunk` processes larger groups of tasks sequentially within each thread.



`parListChunk` `parMap`

Figure 5: Running on 8 threads with chunk size = 128

However, despite the runtime advantage, `parMap` still suffered from **significant GC overhead**, which suggests that the brute-force solver generates too many intermediate results, leading to memory pressure.

In conclusion, the basic brute-force parallel solver demonstrated only limited performance improvements due to **GC overhead** and **load balancing issues**. The best performance was observed with `parMap` and a chunk size of 32, striking a balance between load distribution and memory management. While parallelization improved efficiency compared to sequential execution, further optimization is needed to address garbage collection inefficiencies and dynamically balance workloads to achieve consistent scalability across threads.

3.2.2 Parallel DPLL Static

The DPLL parallel solver utilizes `parMap` and `parListChunk` to distribute subproblems across multiple threads, solving each using the DPLL algorithm. The testing CNF formula consisted of **100 variables**, **50,000 clauses**, and **5 literals per clause**. The solver demonstrated significant performance improvements over the brute-force approach, achieving a peak speedup of 6x with 6 threads. This result highlights the solver's ability to scale efficiently under parallelization while effectively balancing tasks across threads.

The speedup graph shows that performance improves with increasing thread counts, peaking at 6 threads. However, the results are not perfectly linear due to the inherent randomness in selecting variables and literals. Occasionally, a satisfying solution is found early in the search, leading to sudden performance boosts. While this introduces some variability in the observed speedup, it does not detract from the overall scalability and effectiveness of the parallel approach.

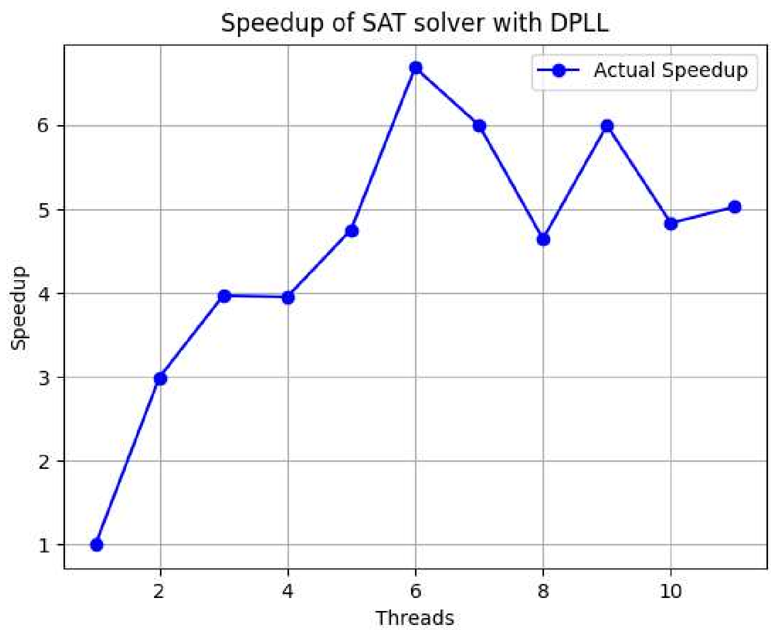


Figure 6: DPLL Speedup - static

We also explored fixing different numbers of variables to divide the search space into smaller, independent subproblems. The runtime and garbage collection times varied depending on the number of fixed variables. Fixing 4 variables resulted in a **runtime of 14.653s** and a **GC time of 3.076s**. All results were obtained using `parMap` with 8 threads.

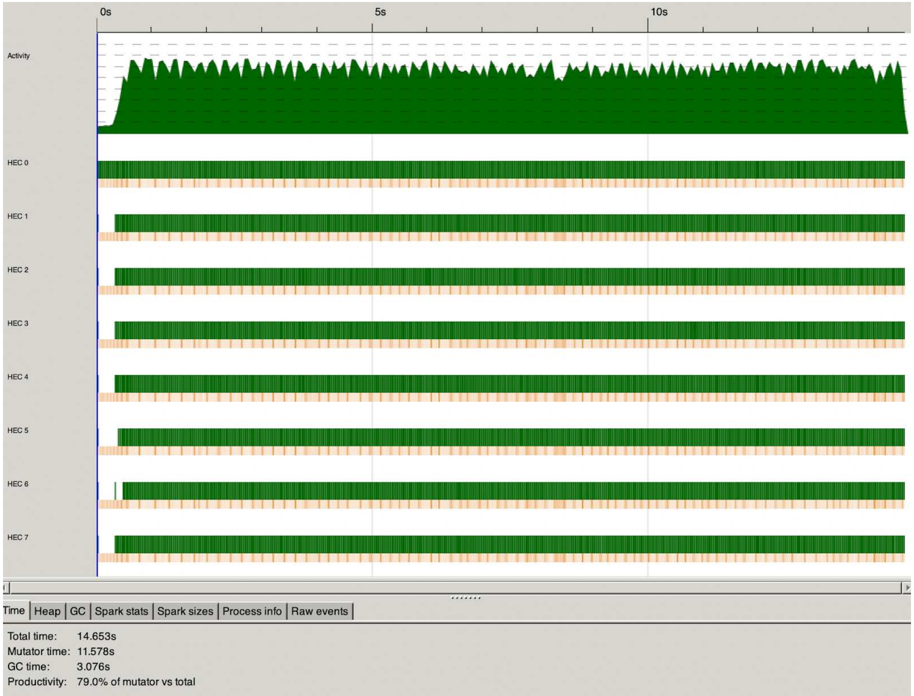


Figure 7: DPLL: Fix 4 variables

Fixing 5 variables provided the best balance, with a **runtime of 14.948s** and the lowest **GC time of 2.961s**.

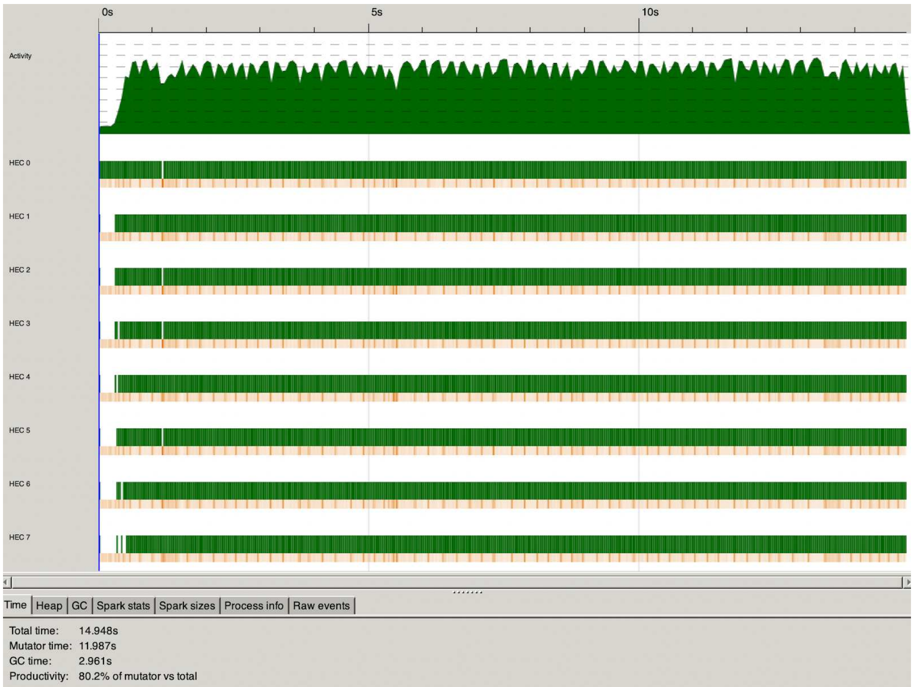


Figure 8: DPLL: Fix 5 variables

Fixing 6 variables increased the **runtime to 29.927s**, with **GC time rising to 6.239s**, as the solver had to manage a larger number of smaller subproblems.

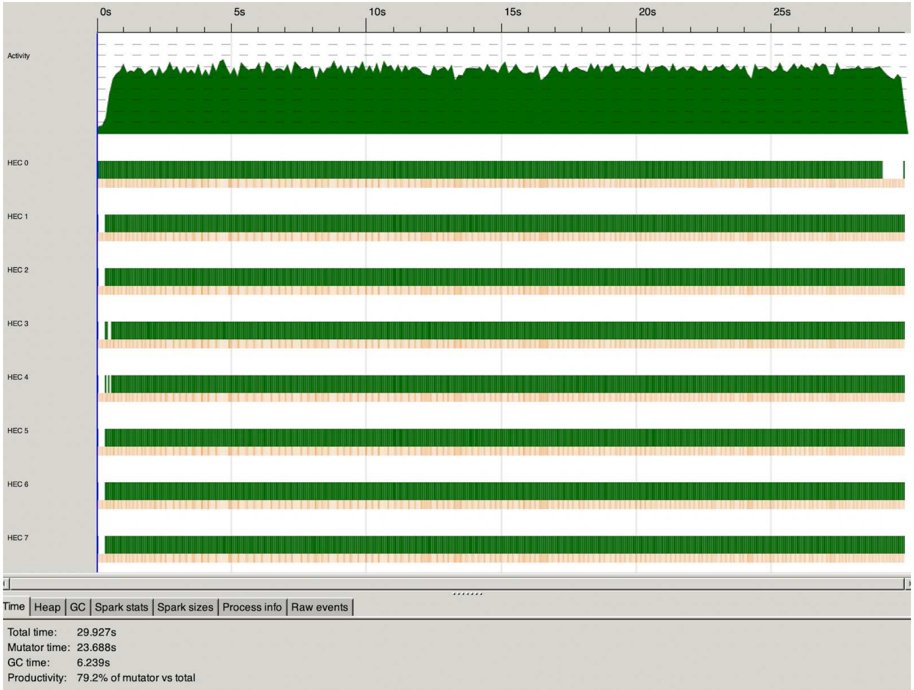
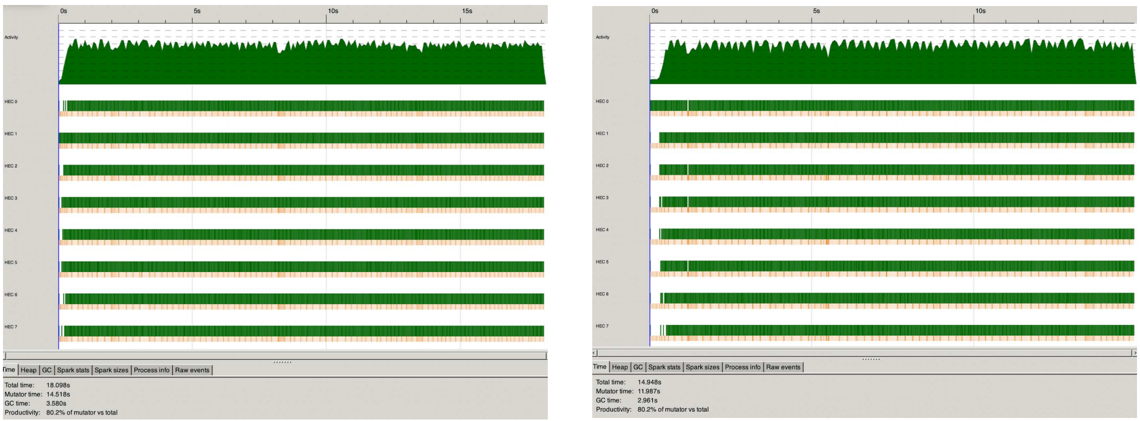


Figure 9: DPLL: Fix 6 variables

This analysis shows that fixing fewer variables increases the chances of finding solutions quickly due to randomness, while fixing too many variables introduces overhead without consistent performance gains.

The Threadscope analysis shows that `parMap` performs better than `parListChunk` for distributing tasks across threads. `parMap` distributes tasks evenly, achieving a **runtime of 14.948s** and **GC time of 2.961s**. Threads work independently, maximizing CPU usage and benefiting from early termination when solutions are found. For `parListChunk` which divides tasks into fixed-size chunks, although threads remain active, the sequential processing of each chunk can delay some threads from moving to new tasks. While the difference isn't significant, it slightly increases runtime compared to `parMap`.



`parListChunk` `parMap`

Figure 10: Running on 8 threads with 5 fixed variables

The results indicate that parMap performed better for parallelized DPLL solver, due to its finer-grained parallelism and reduced overhead compared to chunk-based processing.

3.2.3 Parallel DPLL Queue

The worker queue-based DPLL parallel solver dynamically distributes subproblems across multiple threads using a shared task queue, which ensures that idle threads can pick up remaining tasks dynamically, achieving effective load balancing. Same as previous method, the testing CNF formula consisted of **100 variables**, **50,000 clauses**, and **5 literals per clause**.

According to the speedup graph, there is a significant improvement at 7 threads with a peak of more than 20x speedup.

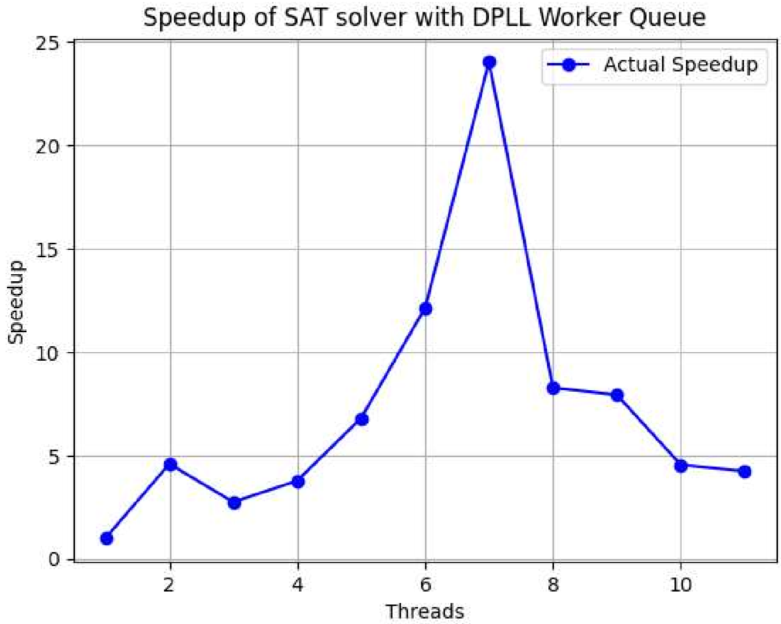


Figure 11: DPLL Speedup - queue

While this performance gain reflects the benefits of dynamic task distribution, it can also be influenced by randomness in variable selection. If the solver finds a satisfying solution early due to a fortunate choice of variables, the computation terminates quickly, leading to an exaggerated speedup. Despite this variability, the results clearly show the worker queue's ability to efficiently distribute work across threads.

The Threadscope analysis also provides additional evidence of the method's efficiency. At the beginning, the first thread performs sequential work to initialize tasks and atomically add them to the shared queue. During this phase, other threads experience a slight delay as they wait for tasks to become available. However, once tasks are distributed, the threads operate consistently with no significant idle time, as reflected in the balanced workload across all threads.

This approach minimizes load imbalance and ensures that all threads remain productive once the queue is populated. The use of `STM` (Software Transactional Memory) enables safe and efficient task synchronization, contributing to the method's overall performance.

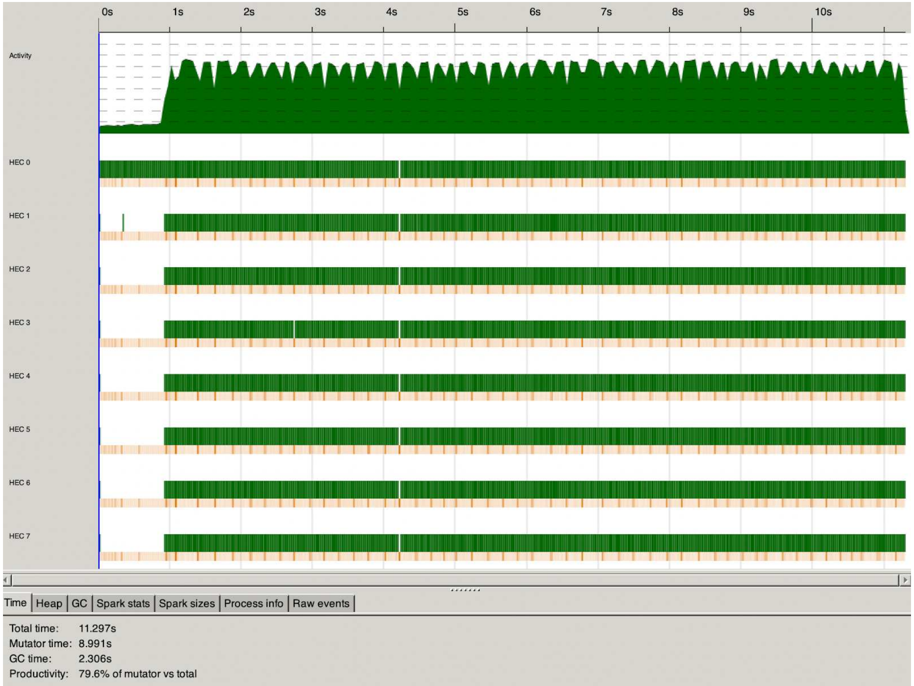


Figure 12: DPLL Queue - threadscope

4 Discussion and Conclusion

4.1 Comparison of Parallel Strategies

Brute-force SAT solver provides a simple baseline by searching all possible variable assignments, but it **scales poorly** due to its **exponential time complexity**. In contrast, the **DPLL solver using static parallelism** (`parMap` and `parListChunk`) improves performance by dividing the search space into fixed subproblems and distributing them across multiple threads. However, this approach can suffer from **load imbalance** when some subproblems are significantly harder to solve than others, **leading to idle threads**. The **worker queue-based DPLL solver** overcomes this limitation by dynamically distributing tasks through a shared queue. This method ensures **better load balancing**, as idle threads can fetch new tasks, and it achieves higher speedup by adapting to varying problem complexity. While static parallelism provides consistent workload distribution upfront, the worker queue approach demonstrates a better **scalability** and **flexibility** in managing computational resources.

4.2 Future Works

There are several optimizations for improving the performance and scalability of the SAT solver that we did not have time to explore. One area is the development of advanced heuristics for branching decisions, such as the Variable State Independent Decaying Sum (VSIDS) heuristic, a dynamic heuristic used in modern SAT solvers, or Most Occurrences in Clauses (MOM), which selects the variable that appears most frequently. These heuristics have the potential to reduce the size of the search space, and improve the time efficiency of the DPLL algorithm.

Another enhancement would be the implementation of Conflict-Driven Clause Learning (CDCL), which is an extension of DPLL that analyzes conflicts to learn new clauses, preventing the solver from revisiting the same conflicts. CDCL also uses non-chronological backtracking, which allows the solver to jump back multiple levels in the decision tree to resolve conflicts more efficiently. To parallelize CDCL, the solver can explore and learn from multiple branches simultaneously.

5 Reference

[1] Martins, R., Manquinho, V., & Lynce, I. (2012). An overview of parallel SAT solving. *Constraints, 17(3),* 304–347. Springer.
[2] Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM, 5(7),* 394–397. ACM New York, NY, USA.
[3] SATLIB - Benchmark Problems

Appendix A: Usage

Code used can be found in this Github Repo.
1. Clone the repository:

```
https://github.com/phoebeww/SAT-Solver.git
cd SAT-Solver
```

2. Install dependencies and compile the project:

```
stack install
stack build
```

3. Run the program:

```
stack run
```

4. Run tests on multiple threads:

```
./test_threads.sh
```

Appendix B: Code Listing

./app/Main.hs

```haskell
-- ./app/Main.hs
module Main (main) where

import Control.DeepSeq (force)
import qualified DPLL.Clause as DClause
import qualified DPLL.DpllSolver as Solver
import qualified DPLL.Literal as DLiteral
import qualified DPLL.ParallelDpll as ParallelSolver
import qualified Data.IntMap as IM
import Data.Maybe (mapMaybe)
import GHC.Conc (getNumCapabilities)
import SatGen (CNF, cnfToDimacs, generateSatisfiableCNF)
import SatBruteForce (solveSATParallel)
import System.Random (newStdGen)
import System.Directory (doesFileExist)

convertCNF :: CNF -> [DClause.Clause]
convertCNF cnf =
  map (\lits -> DClause.mkClause False (map litFromInt lits)) cnf
  where
    litFromInt x
```

```haskell
      | x > 0 = DLiteral.mkLit x False
      | otherwise = DLiteral.mkLit (-x) True

saveCNFAsDimacs :: FilePath -> Int -> CNF -> IO ()
saveCNFAsDimacs path numVars cnf = do
  let dimacs = cnfToDimacs numVars cnf
  writeFile path dimacs
  putStrLn $ "CNF saved in DIMACS format to " ++ path

loadDimacs :: FilePath -> IO (Maybe CNF)
loadDimacs path = do
  content <- readFile path
  let parseClause line =
        let literals = takeWhile (/= 0) . map read . words $ line
         in if null literals then Nothing else Just literals
  let cnf = mapMaybe parseClause . filter (not . null) . filter ((/= 'p') . head) .
              lines $ content
  return $ if null cnf then Nothing else Just cnf

generateAndSaveCNF :: FilePath -> Int -> Int -> Int -> IO ()
generateAndSaveCNF dimacsPath numVars numClauses clauseLen = do
  putStrLn $ "Generating CNF with " ++ show numVars ++ " variables, " ++
              show numClauses ++ " clauses, clause length " ++ show clauseLen
  cnf <- generateSatisfiableCNF numVars numClauses clauseLen
  saveCNFAsDimacs dimacsPath numVars cnf

solveBruteForceCNF :: FilePath -> IO ()
solveBruteForceCNF path = do
  maybeCnf <- loadDimacs path
  case maybeCnf of
    Nothing -> putStrLn "Failed to load CNF from DIMACS file."
    Just cnf -> do
      let numVars = countVariables cnf
      let forcedCnf = force cnf
      case solveSATParallel forcedCnf numVars of
        Nothing -> putStrLn "Brute Force Solver returned UNSATISFIABLE."
        Just result -> do
          putStrLn "Brute Force Solver returned SATISFIABLE."
          putStrLn $ "Satisfying Assignment: " ++ show result

countVariables :: CNF -> Int
countVariables cnf =
  maximum [abs lit | clause <- cnf, lit <- clause]

solveParallelCNF :: FilePath -> IO ()
solveParallelCNF path = do
  maybeCnf <- loadDimacs path
  case maybeCnf of
    Nothing -> putStrLn "Failed to load CNF from DIMACS file."
    Just cnf -> do
      let clauses = force $ convertCNF cnf
      let solver = force $ Solver.newSatSolver {Solver.clauses = clauses}

```

```haskell
      gen <- newStdGen
      case ParallelSolver.parallelSolveOne gen solver of
        Nothing -> putStrLn "Parallel Solver returned UNSATISFIABLE"
        Just result -> do
          putStrLn "Parallel Solver returned SATISFIABLE"
          let solverBindings = Solver.bindings result
          validateSolution cnf solverBindings

solveParallelQueueCNF :: FilePath -> IO ()
solveParallelQueueCNF path = do
  maybeCnf <- loadDimacs path
  case maybeCnf of
    Nothing -> putStrLn "Failed to load CNF from DIMACS file."
    Just cnf -> do
      let clauses = force $ convertCNF cnf
      let solver = force $ Solver.newSatSolver {Solver.clauses = clauses}

      gen <- newStdGen
      numThreads <- getNumCapabilities
      parallelResult <- ParallelSolver.parallelSolveQueue numThreads gen solver
      case parallelResult of
        Nothing -> putStrLn "Parallel Solver returned UNSATISFIABLE"
        Just result -> do
          putStrLn "Parallel Solver returned SATISFIABLE"
          let solverBindings = Solver.bindings result
          validateSolution cnf solverBindings

validateSolution :: CNF -> IM.IntMap Bool -> IO ()
validateSolution cnf bindings = do
  let checkClause clause =
        any
          ( \lit ->
              (lit > 0 && IM.findWithDefault False lit bindings)
                || (lit < 0 && not (IM.findWithDefault False (-lit) bindings))
          )
          clause
  let allSatisfied = all checkClause cnf
  if allSatisfied
    then putStrLn "Solution is valid"
    else putStrLn "Solution is INVALID"

main :: IO ()
main = do
  numThreads <- getNumCapabilities
  putStrLn $ "Number of threads available: " ++ show numThreads

  let dimacsPath = "generated.cnf"
  fileExists <- doesFileExist dimacsPath
  if not fileExists
    then do
      -- if file exists, use the current file. else generate new one
      putStrLn "Generating CNF..."
      -- uncomment for brute force data
```

```
      -- generateAndSaveCNF dimacsPath 25 75 5
      generateAndSaveCNF dimacsPath 100 50000 5
    else putStrLn "CNF file already exists. Using the existing CNF."

  putStrLn "\nSolving Parallel CNF:"
  -- solveBruteForceCNF "generated.cnf"
  -- solveParallelCNF "generated.cnf"
  solveParallelQueueCNF "generated.cnf"
```

./src/SatGen.hs

```
module SatGen
  ( generateSatisfiableCNF,
    cnfToDimacs,
    CNF,
    Clause,
    Literal
  ) where

import System.Random
import Control.Monad
import qualified Data.Set as Set

type Literal = Int
type Clause = [Literal]
type CNF = [Clause]

randomLiteral :: Int -> IO Literal
randomLiteral numVars = do
  var <- randomRIO (1, numVars)
  sign <- randomRIO (False, True)
  return $ if sign then var else -var

generateClauseWithSat :: Int -> Int -> Literal -> IO Clause
generateClauseWithSat numVars len satLit = go (Set.singleton satLit)
  where
    go used
      | Set.size used == len = pure $ Set.toList used
      | otherwise = do
          lit <- randomLiteral numVars
          if Set.member lit used || Set.member (-lit) used
            then go used
            else go (Set.insert lit used)

generateSatisfiableCNF :: Int -> Int -> Int -> IO CNF
generateSatisfiableCNF numVars numClauses clauseLen =
  do
    randVals <- replicateM numVars (randomRIO (0, 1) :: IO Int)
    -- convert from numbers to booleans
    let assignment = map (==1) randVals
    let satisfying = zipWith (\v b -> if b then v else -v) [1..numVars] assignment

    clauses <- replicateM numClauses $ do
      -- pick a random satisfying literal
```

```
      satLit <- (satisfying !!) <$> randomRIO (0, numVars - 1)
      -- genrate the clause using this literal
      generateClauseWithSat numVars clauseLen satLit

    -- remove duplicates
    pure $ Set.toList $ Set.fromList clauses

cnfToDimacs :: Int -> CNF -> String
cnfToDimacs numVars cnf =
  let
    header = "p cnf " ++ show numVars ++ " " ++ show (length cnf)

    clauseToString :: Clause -> String
    clauseToString clause =
        let numbers = map show clause -- convert numbers to strings
            joined = unwords numbers -- join with spaces
        in joined ++ " 0"

    clauseStrings = map clauseToString cnf
    allLines = header : clauseStrings
    in
      unlines allLines
```

./src/SatBruteForce.hs

```
module SatBruteForce (solveSATParallel) where

import Control.Parallel.Strategies
import Control.Monad (replicateM)
import Control.Applicative (Alternative(..))
import Data.Maybe (fromJust)
import Data.List.Split (chunksOf)
import SatGen (CNF, Clause, Literal)

type Assignment = [(Int, Bool)]

generateAllAssignments :: Int -> [[Assignment]]
generateAllAssignments n =
  let allAssignments = [ zip [1..n] bools | bools <- replicateM n [False, True] ]
      chunkSize = 128
  in chunksOf chunkSize allAssignments

evaluateLiteral :: Assignment -> Literal -> Bool
evaluateLiteral assignment lit =
  let variable = abs lit
      value = fromJust (lookup variable assignment)
  in if lit > 0 then value else not value

evaluateClause :: Assignment -> Clause -> Bool
evaluateClause assignment clause = any (evaluateLiteral assignment) clause

evaluateCNF :: Assignment -> CNF -> Bool
evaluateCNF assignment cnf = all (evaluateClause assignment) cnf

```

```haskell
evaluateChunk :: CNF -> [Assignment] -> Maybe Assignment
evaluateChunk cnf assignments =
    findFirstSatisfying assignments
  where
    findFirstSatisfying [] = Nothing
    findFirstSatisfying (assign:rest)
        | evaluateCNF assign cnf = Just assign
        | otherwise = findFirstSatisfying rest

solveSATParallel :: CNF -> Int -> Maybe Assignment
solveSATParallel cnf numVars =
  let chunks = generateAllAssignments(numVars)
      results = parMap rdeepseq (evaluateChunk cnf) chunks
  in foldr (<|>) Nothing results
```

./src/DPLL/Literal.hs

```haskell
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}

module DPLL.Literal
  ( Var,
    var_Undef,
    Lit (..),
    lit_Undef,
    lit_Error,
    mkLit,
    neg,
    sign,
    var,
    index,
    toLit,
    unsign,
    idLit,
    toDimacs,
  )
where

import Data.Bits (complement, shiftR, xor, (.&.))
import GHC.Generics (Generic)
import Control.DeepSeq (NFData)

type Var = Int

var_Undef :: Int
var_Undef = -1

data Lit = Lit {x :: Int}
  deriving (Eq, Show, Generic, NFData)

lit_Undef :: Lit
lit_Undef = Lit (2 * var_Undef)

lit_Error :: Lit
```

```haskell
lit_Error = Lit (2 * var_Undef + 1)

mkLit :: Var -> Bool -> Lit
mkLit v sgn = Lit ((v + v) + if sgn then 1 else 0)

neg :: Lit -> Lit
neg p = Lit (x p `xor` 1)

sign :: Lit -> Bool
sign p = x p .&. 1 == 1

var :: Lit -> Int
var p = x p `shiftR` 1

index :: Lit -> Int
index p = x p

toLit :: Int -> Lit
toLit i = Lit i

unsign :: Lit -> Lit
unsign p = Lit (x p .&. complement 1)

idLit :: Lit -> Bool -> Lit
idLit p sgn = Lit (x p `xor` (if sgn then 1 else 0))

toDimacs :: Lit -> Int
toDimacs p = if sign p then -(var p) - 1 else var p + 1

```

./src/DPLL/Clause.hs

```haskell
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}

module DPLL.Clause
  ( Clause (..),
    mkClause,
    clauseSize,
    getLit,
    isLearnt,
    setActivity,
    getActivity,
  )
where

import DPLL.Literal (Lit)
import GHC.Generics (Generic)
import Control.DeepSeq (NFData)

data Clause = Clause
  { literals :: [Lit],
    learnt :: Bool,
    activity :: Maybe Float
```

```haskell
  }
  deriving (Show, Eq, Generic, NFData)

mkClause :: Bool -> [Lit] -> Clause
mkClause isLearned ps = Clause {literals = ps, learnt = isLearned, activity = Nothing}

clauseSize :: Clause -> Int
clauseSize clause = length (literals clause)

getLit :: Clause -> Int -> Maybe Lit
getLit clause i
  | i >= 0 && i < clauseSize clause = Just (literals clause !! i)
  | otherwise = Nothing

isLearnt :: Clause -> Bool
isLearnt = learnt

setActivity :: Clause -> Float -> Clause
setActivity clause act = clause {activity = Just act}

getActivity :: Clause -> Maybe Float
getActivity = activity

```

./src/DPLL/DpllSolver.hs

```haskell
module DPLL.DpllSolver (
    SatSolver(..),
    newSatSolver, isSolved,
    selectBranchVar, solve,
    guess
) where

import Data.Maybe (mapMaybe)
import DPLL.Clause
import DPLL.Literal
import qualified Data.IntMap as IM
import Control.Applicative (Alternative(..))
import Data.List (sortBy)
import Control.DeepSeq (NFData, rnf)

data SatSolver = SatSolver
  { clauses :: ![Clause],            -- Force strict evaluation of clauses
    bindings :: !(IM.IntMap Bool)   -- Force strict evaluation of bindings
  }
  deriving (Show, Eq)

instance NFData SatSolver where
    rnf solver = rnf (clauses solver) `seq` rnf (bindings solver)

newSatSolver :: SatSolver
newSatSolver = SatSolver [] IM.empty

selectBranchVar :: SatSolver -> Var
```

```haskell
selectBranchVar solver =
    var $ head $ literals $ head $ sortBy shorterClause (clauses solver)

isSolved :: SatSolver -> Bool
isSolved = null . clauses

solve :: (Monad m, Alternative m) => SatSolver -> m SatSolver
solve solver =
    maybe empty solveRecursively (simplify solver)

solveRecursively :: (Monad m, Alternative m) => SatSolver -> m SatSolver
solveRecursively solver
    | isSolved solver = pure solver
    | otherwise = do
        let varToBranch = selectBranchVar solver
        branchOnUnbound varToBranch solver >>= solveRecursively

branchOnUnbound :: (Monad m, Alternative m) => Var -> SatSolver -> m SatSolver
branchOnUnbound name solver =
    guessAndRecurse (mkLit name True) solver
    <|>
    guessAndRecurse (mkLit name False) solver

guessAndRecurse :: (Monad m, Alternative m) => Lit -> SatSolver -> m SatSolver
guessAndRecurse lit solver = do
    case guess lit solver of
        Nothing -> empty -- Conflict detected, backtrack
        -- Continue solving recursively
        Just simplifiedSolver -> solveRecursively simplifiedSolver

guess :: Lit -> SatSolver -> Maybe SatSolver
guess lit solver =
    let updatedBindings = IM.insert (var lit) (not (sign lit)) (bindings solver)
        updatedClauses = mapMaybe (filterClause lit) (clauses solver)
    in simplify $ solver { clauses = updatedClauses, bindings = updatedBindings }

simplify :: (Monad m, Alternative m) => SatSolver -> m SatSolver
simplify solver = do
    case findUnitClause (clauses solver) of
        Nothing -> pure solver
        Just lit -> do
            let updatedSolver = solver { bindings = IM.insert (var lit) (not (sign lit))
                                                (bindings solver) }
            case propagate lit (clauses updatedSolver) of
                Nothing -> empty
                Just updatedClauses ->
                    simplify $ updatedSolver { clauses = updatedClauses }

propagate :: Lit -> [Clause] -> Maybe [Clause]
propagate lit inputClauses =
    let updatedClauses = mapMaybe (processClause lit) inputClauses
    in if any (null . literals) updatedClauses
       then Nothing
```

```haskell
        else Just updatedClauses

findUnitClause :: [Clause] -> Maybe Lit
findUnitClause [] = Nothing
findUnitClause (c:cs)
    | clauseSize c == 1 = Just (head (literals c))
    | otherwise = findUnitClause cs

processClause :: Lit -> Clause -> Maybe Clause
processClause lit clause
    | lit `elem` literals clause = Nothing
    | neg lit `elem` literals clause =
        let newLits = filter (\l -> l /= neg lit) (literals clause)
        in if null newLits
           then Just $ Clause [] (learnt clause) (activity clause)
           else Just $ Clause newLits (learnt clause) (activity clause)
    | otherwise = Just clause

filterClause :: Lit -> Clause -> Maybe Clause
filterClause lit clause
    | lit `elem` literals clause = Nothing
    | neg lit `elem` literals clause =
        Just $ Clause (filter (\l -> l /= neg lit) (literals clause)) (learnt clause)
                                    (activity clause)
    | otherwise = Just clause

shorterClause :: Clause -> Clause -> Ordering
shorterClause c1 c2 = compare (clauseSize c1) (clauseSize c2)

```

./src/DPLL/ParallelDpll.hs

```haskell
module DPLL.ParallelDpll (
    SatSolver(..),
    parallelSolveOne,
    parallelSolveQueue,
    parallelSolveDynamicQ
) where

import Control.Concurrent
import Control.Concurrent.STM
import Control.Parallel.Strategies
import Data.Maybe (mapMaybe, listToMaybe)
import qualified Data.IntMap.Strict as IM
import qualified Data.IntSet as IS
import System.Random (StdGen, randomRs)
import Control.Monad (replicateM_, foldM)
import DPLL.DpllSolver
import DPLL.Literal
import DPLL.Clause

parallelSolveDynamicQ :: Int -> StdGen -> SatSolver -> IO (Maybe SatSolver)
parallelSolveDynamicQ  numThreads gen solver = do
    taskQueue <- newTQueueIO
```

```haskell
    resultsVar <- newEmptyMVar

    let vars = selectRandomVars gen solver
    let subproblems = generateSubproblems vars solver
    atomically $ mapM_ (writeTQueue taskQueue) subproblems

    replicateM_ numThreads $ forkIO $ worker taskQueue resultsVar
    takeMVar resultsVar
  where
    worker taskQueue resultsVar = do
        maybeTask <- atomically $ tryReadTQueue taskQueue
        case maybeTask of
            Nothing -> return ()  -- No more work
            Just subproblem -> do
                case solve subproblem of
                    Just solution -> putMVar resultsVar (Just solution)
                    Nothing -> do
                        -- find & add new subproblem dynamically
                        let newSubproblems = splitSubproblem solver
                        atomically $ mapM_ (writeTQueue taskQueue) newSubproblems
                        worker taskQueue resultsVar  -- Continue working

splitSubproblem :: SatSolver -> [SatSolver]
splitSubproblem solver =
    let variable = selectBranchVar solver
    in [ solver { bindings = IM.insert variable True (bindings solver) },
         solver { bindings = IM.insert variable False (bindings solver) }
       ]

parallelSolveQueue :: Int -> StdGen -> SatSolver -> IO (Maybe SatSolver)
parallelSolveQueue numThreads gen solver = do
    taskQueue <- newTQueueIO -- shared work queue
    resultsVar <- newEmptyMVar -- result

    let vars = selectRandomVars gen solver
    let subproblems = generateSubproblems vars solver

    -- add the subproblems to the queue. *atomically* used for atomic transaction
    atomically $ mapM_ (writeTQueue taskQueue) subproblems

    -- start parallel processing
    replicateM_ numThreads $ forkIO $ worker taskQueue resultsVar
    takeMVar resultsVar -- blocked until a result is added
  where
    worker taskQueue resultsVar = do
        maybeTask <- atomically $ tryReadTQueue taskQueue -- read a task
        case maybeTask of
            Nothing -> return () -- exit with no left work
            Just subproblem -> do
                let result = solve subproblem
                case result of
                    Just solution -> putMVar resultsVar (Just solution)
                    Nothing -> worker taskQueue resultsVar
```

```haskell

parallelSolveOne :: StdGen -> SatSolver -> Maybe SatSolver
parallelSolveOne gen solver =
    let vars = selectRandomVars gen solver
        subproblems = generateSubproblems vars solver
        results = parMap rdeepseq solve subproblems
    in listToMaybe (mapMaybe id results) -- return first solution

selectRandomVars :: StdGen -> SatSolver -> [Var]
selectRandomVars gen solver =
    let allVars = IS.toList $ IS.fromList
                  [var lit | clause <- clauses solver, lit <- literals clause]
        indices = take 5 $ randomRs (0, length allVars - 1) gen -- take 5 random vars
    in map (allVars !!) indices

generateSubproblems :: [Var] -> SatSolver -> [SatSolver]
generateSubproblems vars solver =
    -- some assignments may fail due to conflicts, filter them
    mapMaybe (`applyAssignment` solver) (generateAssignments vars)

generateAssignments :: [Var] -> [[Lit]]
generateAssignments vars =
    [[mkLit v val | (v, val) <- zip vars vals] | vals <- sequence (replicate
        (length vars) [True, False])]

applyAssignment :: [Lit] -> SatSolver -> Maybe SatSolver
applyAssignment lits baseSolver =
    foldM (\solver lit -> guess lit solver) baseSolver lits
```