SAT Solver Parallelization (tsp-sat)
Final Presentation

COMS W 4995: Parallel Functional Programming

Yixuan Li
Phoebe Wang
Jiaqian Li

Dec 17, 2024

Columbia | Engineering
The Fu Foundation School of Engineering and Applied Science

Team

Team Members	Contribution Highlights
Phoebe Wang	Brute force SAT solver optimization, Worker queue implementation for DPLL.
Jiaqian Li	DPLL implementation and optimization.
Yixuan Li	Brute force SAT solver implementation and DPLL parallelization.

Introduction

What is SAT Solving?

- whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.

$$(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2 \lor x_3) \land \neg x_1$$

If we choose `x1 = FALSE, x2 = FALSE`, and `x3` arbitrarily

⇩

`(FALSE ⋁ ¬FALSE) ⋀ (¬FALSE ⋁ FALSE ⋁ x3) ⋀ ¬FALSE`

Real world applications

- Software Verification: validate program correctness.
- Machine Learning: check if a neural network behaves as expected under specific conditions.
- Constraint Satisfaction Problems (CSPs): find winning strategies in games like Sudoku, or chess.

Basic SAT Solver Implementation

The basic approach to solving a SAT problem is to **enumerate all possible assignments** for the variables in the formula.

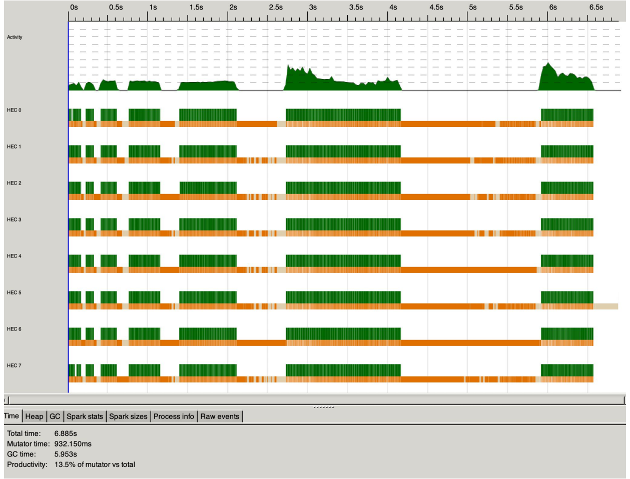
- For a problem with **`n`** variables, this involves testing all **`2^n`** possible assignments to see if any satisfies the formula.

Algorithm:

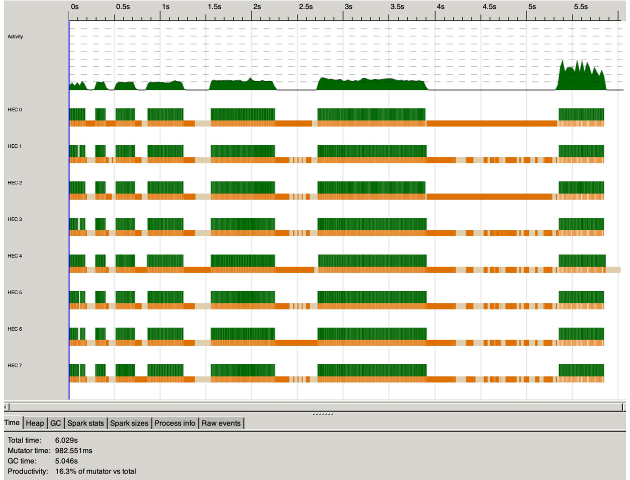
- Generate all possible assignments for the variables.
- Divide assignments into small chunks. (`chunkSize = 16/32/64`)
- For each chunk:
  - Evaluate all assignments in the chunk.
  - Return the first satisfying assignment, if found.
- Combine results from all chunks in parallel. (`parListChunk / parMap`)
  - We use `rdeepseq` to prevent lazy evaluation from leaving unevaluated thunks in memory

Basic SAT Solver Result - Chunk Size

Running on 8 threads with `map (evaluateChunk cnf) chunks `using` parListChunk 4 rdeepseq`:



Chunk size = 16
Time: 6.885s
GC: 5.953s

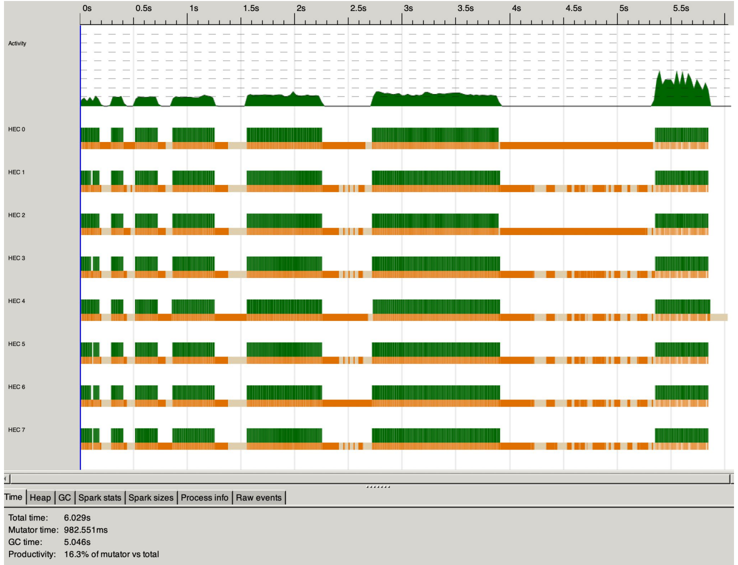


Chunk size = 32
Time: 6.029s
GC: 5.048s

Chunk size = 64
Time: 7.525s
GC: 2.971s

Basic SAT Solver Result - Parallel Strategy

Running on 8 threads with `chunkSize = 128`:



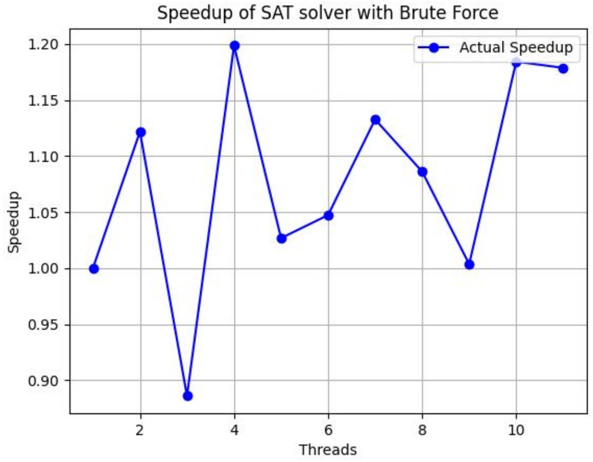
`parListChunk`
Time: 6.029s
GC: 5.048s

`parMap`
Time: 5.267s
GC: 4.078s

Basic SAT Solver Result - Speedup

Running on 1 to 11 threads with `parMap` and 128 chunks:

- Total number of variables: 25
- Total number of clauses: 75
- Number of literals per clause: 5



Analysis of all graphs:

Threadscope Graph

- Pro: The parallel workload is fairly well distributed.
- Con: Garbage collection (GC) dominates the runtime.

Speedup Graph

- The speedup fluctuates between 1-11 (peak speedup of **1.2x** at 4 threads).
- No obvious improvement running with parallelization.
    - Potential reason: High GC Overhead

DPLL For SAT Solver

Main issues with the basic implementation

- Infeasible to handle large inputs (time complexity: `O(2^n)`)
- Memory-intensive (unnecessary check once assignment is invalid)

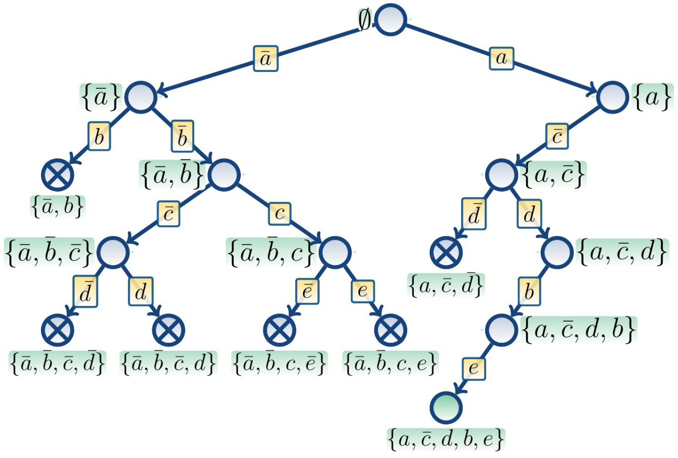
Sequential DPLL Algorithm:

Base Cases:

If the formula has no clauses → satisfiable

If the formula contains an empty clause → not satisfiable.

1. Pick a literal not yet assigned and guess its value (e.g., TRUE).
2. Recursively find other variables in the formula after assigning the decision literal.
3. If the formula becomes unsatisfiable, backtrack and try the opposite value.



DPLL Parallelization

Unit Propagation:
- If a clause has a single literal, assign it a value that satisfies it.
- Simplify the formula by removing satisfied clauses and the negated literal from others.

$$(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor x_3 \lor \neg x_4) \land (\neg x_3)$$

From (`not x3`), assign `x3 = False`

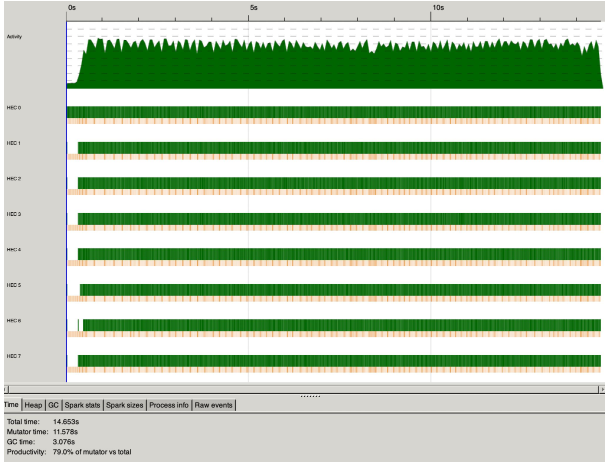
$$(x_1 \lor \neg x_2) \land (\neg x_1) \land (x_2 \lor \neg x_4).$$

Time Complexity: Worse case `O(2^n)` but unit propagation often simply the formula.

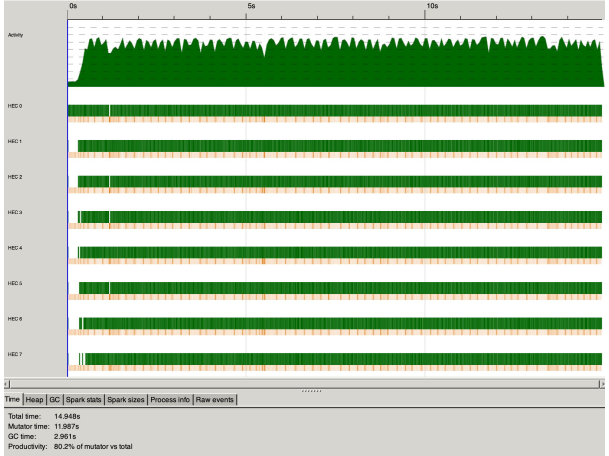
Parallel Approach 1 (with inspiration from brute force):
- Randomly select a small subset of variables (`k = 4/5/6`) from the formula.
- Create all possible combinations of truth assignments (`2^k = 16/32/64` combinations) for the selected variables.
- Solve the generated subproblems parallelly (`parMap`)

DPLL Parallelization Results

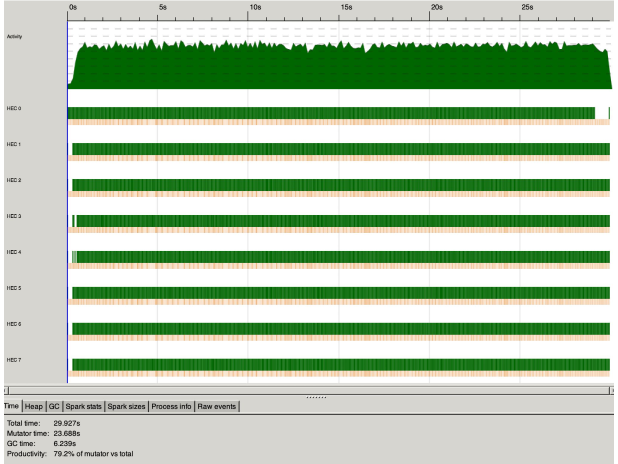
Running on 8 threads using `parMap` fixing `4/5/6` num variables:



Fix 4 variables
Time: 14.653s
GC: 3.076s



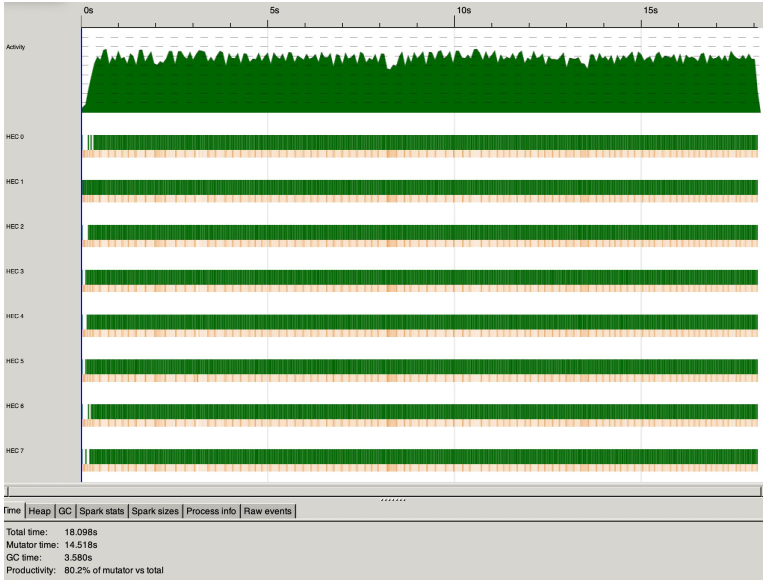
Fix 5 variables
Time: 14.948s
GC: 2.961s

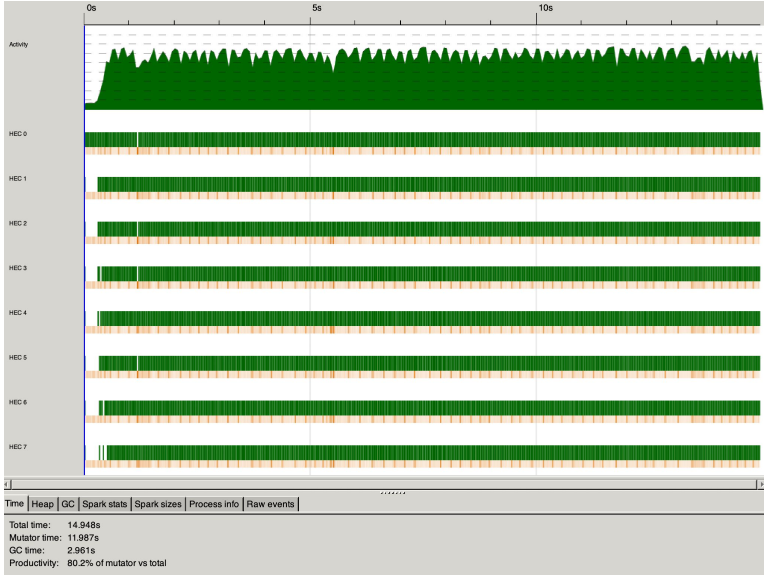


Fix 6 variables
Time: 29.927s
GC: 6.239s

DPLL Parallelization Results

Running on 8 threads using `parMap` and `parListChunk` fixing `5` variables:





`parListChunk`

`Chunk = subproblems/numThreads`

Time: 18.098s

GC: 3.580s

`parMap`

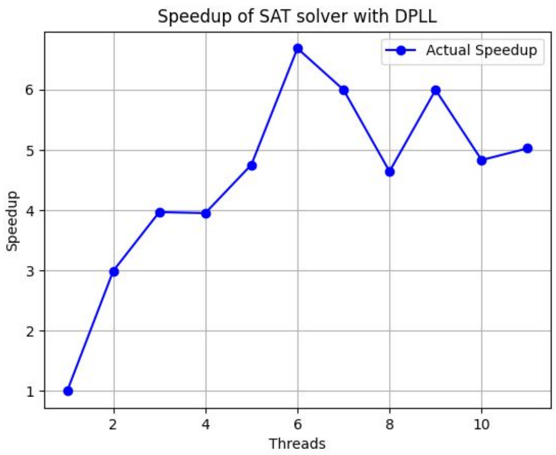
Time: 14.948s

GC: 2.961s

DPLL Parallelization Results

Running on 1 to 11 threads with `parMap` and fixing 5 variables:

- Total number of variables: 100
- Total number of clauses: 50,000
- Number of literals per clause: 5



Analysis of all graphs:

<u>Threadscope Graph</u>

- All threads have consistent workload distribution.
- Garbage collection (GC) has reduced significantly. (productivity increased to 80%)

<u>Speedup Graph</u>

- The speedup is better compared to the brute force approach, and peaks at 6 threads with more than 6x speedup.

<u>Efficiency:</u>

- Can handle large inputs.
- Takes 24.845ms (previously 5.267s) on 8 threads on the 25 numVars, 75 numClauses.

DPLL Parallelization With Worker Queue

Parallel Approach 2: Shared Work Queue

Task Queue

- A shared queue is used to store and manage subproblems
- Threads pull tasks (subproblems) from the queue
- Synchronization is handled using `Control.Concurrent.STM`

Steps

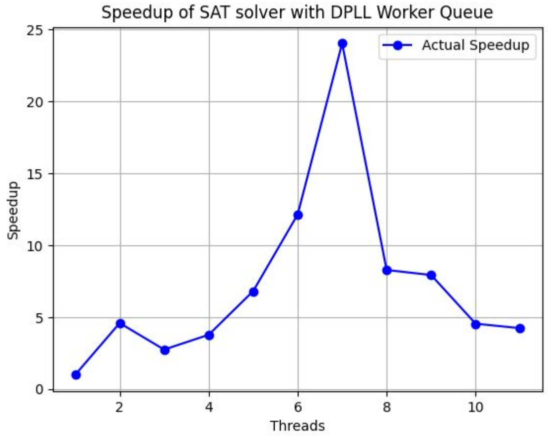
- Generate initial subproblems by selecting a small subset of variables
- Add resulting subproblems to the task queue
- Each thread fetches a task from the queue and attempts to solve
- Threads keep working until the queue is empty or a solution is found

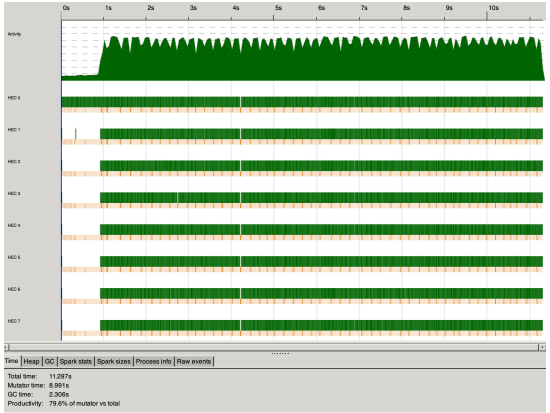
Time Complexity

- The time complexity remains `O(2^n)` in the worst case, but the worker queue reduces idle time and improves practical performance by dynamically balancing the workload across threads

DPLL Parallelization With Worker Queue Results

Running on 8 threads with worker queue with the same test data as the previous test:





Analysis of both graphs:

<u>Threadscope Graph</u>

- All threads have consistent workload distribution.
- More sequential work at the beginning on the 1st thread.

<u>Speedup Graph</u>

- The speedup is better compared to the previous DPLL approach, and peaks at 7 threads with more than 20x speedup.

SAT Solver Test Data Generator

<u>Input</u>

- `numVars`: Number of variables in the formula.
- `numClauses`: Number of clauses in the formula.
- `clauseLen`: Number of literals per clause.

1. Generate a random truth assignment for all variables.
2. Generate clauses
   a. For each clause, pick a random satisfying literal from the assignment
   b. Creates a clause of length `clauseLen` containing the satisfying literal and other random literals
3. Remove the duplicate clauses

<u>Output</u>: a list of clauses

`numVars = 4, numClauses = 3, clauseLen = 3`

⇩

`Assignment: [T, F, T, F].`
`Literals: [1, -2, 3, -4].`

⇩

Clause 1: `(1 ∨ 2 ∨ -3)` (includes satisfying literal 1).

Clause 2: `(-2 ∨ 4 ∨ -1)` (includes satisfying literal -2).

Clause 3: `(3 ∨ -4 ∨ 2)` (includes satisfying literal 3).

Future Work

Heuristics for Variable Selection

- <u>Current limitation</u>: we chooses variables in a random manner.
- Implement advanced heuristics such as:
  - Most Occurrences in Clauses (MOM): Select the variable that appears most frequently.
  - VSIDS (Variable State Independent Decaying Sum): A dynamic heuristic used in modern SAT solvers.

Conflict-Driven Clause Learning (CDCL)

- <u>Current limitation</u>: If a branch fails, go back to the previous level.
- Benefits of CDCL:
  - CDCL analyzes conflicts to determine why the conflict occurred and generates a learned clause.
  - Jump back multiple levels to the cause of the conflict.

Thank You!