SATlab

AE6102 - Parallel Scientific Computing and Visualization

Student 1 Name: Anmol Saraf
Roll No. 1: 200070007
Instructor: Prof. Prabhu Ramachandran

Student 2 Name: Nikhil Kaniyeri
Roll No. 2: 200070050
Due Date: 1^{th} May, 2023

Abstract

In computer science, the boolean satisfiability problem or the SAT problem is the procedure of automated theorem proving that can simplify a set of clauses. We propose to create a SAT Solver that uses classical methods like DPLL and CPCL combined with other tricks parallely to create a faster unit solver. We also intend to apply this solver on use cases like graph network problems (circuit et al.) as well as visualization by implementing GUI for games like Sudoku and Minesweeper. Finally, we will attempt cryptanalysis on hashing by using inversion attacks on secure hash functions.

1 Introduction

The boolean satisfiability problem, or the SAT problem, is the problem of determining whether a solution exists that satisfies a given boolean formula. For example, the formula `A and B` has a valid solution, A being true and B being true, whereas `A and not A` has no valid solution. It is a significant problem in computer science and logic, as it is an NP-complete problem, in a group of various optimization problems. We propose to create a parallel solver for the above problem.

2 Tools used

We use mpi2py to communicate between multiple parallel threads while running the unit propagation step. [Refer to methods of SAT solving section.] We use numba to accelerate the array computation. The sudoku was displayed using pygame.

3 Outline

- We have implemented a SAT solver from scratch using the DPLL Algorithm.
- We have an implementation of the same SAT solver that does unit propagation parallelly for speedup. We had most of our difficulties here, especially in the initial splitting of tasks across all threads and then updating with the information gained till now.
- We have a working proof of our SAT solver working on Sudoku. We did not write the CNFs ourselves, the clause generation logic was taken from references.
- We proposed to attempt cryptanalysis on a ciphertext, we were not able to achieve this goal from our original proposal.

4 Showcase

We have results comparing the normal single thread execution and multithread execution.

```
nx6xe23@cerberus:~/github-repos/SATlab/dpll$ python3 dpll.py
True
0.005504131317138672
[-1, -2, -3, 4, -5, -6, 7, -8, 9, -10, -11, -12, -13, 14, -15, -16, 17, -18, -19, -20, -21, 22, -23, -24, -25, -
26, 27, -28, -29, -30, -31, 32, -33, -34, 35, -36, -37, -38, -39, 40, -41, 42, -43, -44, 45, -46, -47, -48, -49,
 50, -51, -52, 53, -54, -55, -56, -57, -58, -59, 60, -61, -62, 63, -64]
```

Figure 1: Single Thread Execution

```
nx6xe23@cerberus:~/github-repos/SATlab/dpll$ mpiexec -n 3 python3 dpll_mpi.py
0.0034286975860595703
[-1, -2, -3, 4, -5, -6, 7, -8, 9, -10, -11, -12, -13, 14, -15, -16, 17, -18, -19, -20, -21, 22, -23, -24, -25, -
26, 27, -28, -29, -30, -31, 32, -33, -34, 35, -36, -37, -38, -39, 40, -41, 42, -43, -44, 45, -46, -47, -48, -49,
 50, -51, -52, 53, -54, -55, -56, -57, -58, -59, 60, -61, -62, 63, -64]
```

Figure 2: MPI run with n=3

```
nx6xe23@cerberus:~/github-repos/SATlab/dpll$ mpiexec -n 7 python3 dpll_mpi_7.py
[-1, -2, -3, 4, -5, -6, 7, -8, 9, -10, -11, -12, -13, 14, -15, -16, 17, -18, -19, -20, -21, 22, -23, -24, -25, -
26, 27, -28, -29, -30, -31, 32, -33, -34, 35, -36, -37, -38, -39, 40, -41, 42, -43, -44, 45, -46, -47, -48, -49,
 50, -51, -52, 53, -54, -55, -56, -57, -58, -59, 60, -61, -62, 63, -64]
0.0037479400634765625
```

Figure 3: MPI run with n=7

5 Deliverables

What we achieved:

- A parallel SAT solver with a speedup of around 2.
- Working example on a 4x4 Sudoku

6 Timeline

Time Duration	Task to be Completed
26 Feb - 12 Mar	Leaning and implementing algorithms parallelly
13 Mar - 10 Apr	Implementing Message Passing, to help concurrent threads from current executing threads
11 Apr - 26 Apr	Usage of solver on benchmarks; creating Fast Sudoku/Minesweeper with GUI

7 Methods of SAT solving

7.1 Unit Propogation

This method is also called the one-literal rule, since it is based on unit clauses. Because each clause needs to be satisfied, we know that this literal is true. Other clauses that contain this unit literal get simplified by the application of a few rules as described below:

- Any clause containing the literal is removed. (The clause is satisfied *as-is.*)

- In every clause that contains the negation of the literal, the literal is deleted. (The negation cannot contribute in it being satisfied.)

The two most common and efficient algorithms that build on this idea for solving SAT problems are known as the **Davis–Putnam–Logemann–Loveland** (DPLL) algorithm and the **Conflict-Driven Clause Learning** (CPCL) algorithm. These are defined as follows,

7.2 Davis–Putnam–Logemann–Loveland

DPLL is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. if a boolean logic is a conjunction of one or more clauses. The DPLL algorithm for solving a SAT problem is as follows,

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

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

This is a parallelly implementable algorithm but there is no learning from the wrong guesses made for the variables. The following algorithm CPCL takes account for that.

7.3 Conflict-Driven Clause Learning

The CPCL algorithm follows the same methodology as the DPLL algorithm but the back-jumping in CPCL is non-chronological. The main algorithm is as given below,

1. Select a variable and assign True or False. This is called decision state. Remember the assignment.
2. Apply Boolean constraint propagation (unit propagation).
3. Build the implication graph.
4. If there is any conflict:
   (a) Find the cut in the implication graph that led to the conflict.
   (b) Derive a new clause which is the negation of the assignments that led to the conflict.
   (c) Non-chronologically backtrack ("back-jump") to the appropriate decision level, where the first-assigned variable involved in the conflict was assigned.

5. Otherwise continue from step 1 until all variable values are assigned.

This updates the memory but reduces the steps needed to solve the problem. This is a complex time and space trade-off algorithm.

8 Links

Github Repository

https://www.github.com/kaniyeri/SATlab

References

https://en.wikipedia.org/wiki/DPLL_algorithm
https://en.wikipedia.org/wiki/Conflict-driven_clause_learning
https://en.wikipedia.org/wiki/Implication_graph
https://eprint.iacr.org/2006/254.pdf