Satisfiability Solver using Parallel DPLL

Cassius Garcia, Varun Iyengar, and Harrison Muller

***Abstract*—The Boolean satisfiability problem is a major area of research in the realm of logic synthesis, as it indicates whether there exists a set of implications for which a Boolean formula is satisfied. The boolean satisfiability problem is addressed through the use of satisfiability solvers (SAT solvers), which utilize various algorithms to determine the satisfiability of a function. One of these algorithms is the Davis-Putnam-Logemann-Loveland Algorithm (DPLL), which is a smarter and faster version of many simple SAT solvers currently in use. In this project, we have constructed a DPLL algorithm using Python3, with the addition of parallelism granted by the threading library to speed up the algorithm for very large POS formulas in conjunctive norm form (CNF). Our team ran the algorithm on three benchmarks and noticed that for smaller indexes the parallel DPLL works slower than existing single-core algorithms found online. However, we anticipate that for larger CNFs the parallel DPLL will run exponentially faster than traditional methods.**

***Index Terms*—satisfiability, SAT solver, DPLL, CNF, POS, clause, literal, unit clause, unit propagation, backtracking, BCP, threads, parallelism, CPU, Python3, etc.**

I. Introduction

SATISFIABILITY is a key component of logic synthesis due to its role in determining if there exists a set of literal implications that can make a boolean function solve to true. If a combination does exist, the function is said to be satisfiable. If no combination exists, the function is unsatisfiable. Currently no agreed-upon algorithm or set therein is capable of solving all satisfiability problems, but there exist numerous methods known as satisfiability solvers (SAT solvers) which seek to prove whether or not boolean functions are satisfiable and return the implication that make them as such. SAT solvers can be implemented in numerous ways, but for this project, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm was utilized. A diagram of the DPLL algorithm as presented in the lecture is provided in Figure 1.

```
DPLL(set_of_clauses )
    // do BCP
    while (set_of_clauses contains a unit clause due to literal L) {
        Simplify set_of_clauses by setting variable for L to its required value in all clauses
    }
    If (set_of_clauses is all "1" clauses now)
        return ( SAT ) // you have simplified every clause to be "1"
    If (set_of_clauses contains a clause that evals to "0")
        return (UNSAT) // this is a conflict, this set of var assignment doesn't satisfy
    // must recurse
    Heuristically choose an unassigned variable x and heuristically choose a value v
        If ( DPLL( set_of_clauses = simplified by setting x=v ) == SAT )
            return SAT
        else return DPLL( set_of_clauses = simplified by setting x= ¬v )
```

Fig. 1. Basic DPLL Algorithm Pseudo-Code [**?**]

This algorithm works by first performing boolean constraint propagation (BCP), which simplifies the provided CNF formula in terms of a variable L by setting it to its required values in all clauses. While the CNF formula still contains unit clauses due to variable L, BCP continues until no more implications remain. The DPLL algorithm follows this up by checking if all clauses in the CNF formula evaluate to one or if a clause is found to evaluate to zero. These checks return satisfiable or unsatisfiable respectfully. If neither comparison returns, a splitting variable is chosen using some heuristic and DPLL is executed recursively using the splitting variable. If this recursion does not work, the algorithm reruns the recursion, this time using the negated variable. An example of DPLL can be seen in Figure 2. In this example, the function splits on A since no unit clauses currently exist. The function is recursed upon and the decision is taken to set the value to 0. This continues for other variables B and C, which are split and recursed on while also eliminating any resulting unit clauses using unit propagation. However, the solution is unsatisfied after choosing 0 for C, so new variable assignments need to be made. Bakctracking occurs through recursion until we return to A and try the positive version of the variable. Recursing further, we are able to find that the far right solution is satifiable, allowing up to return. Unlike this example, however, we recurse on the positive variable first, effectively solving the problem in two splits.

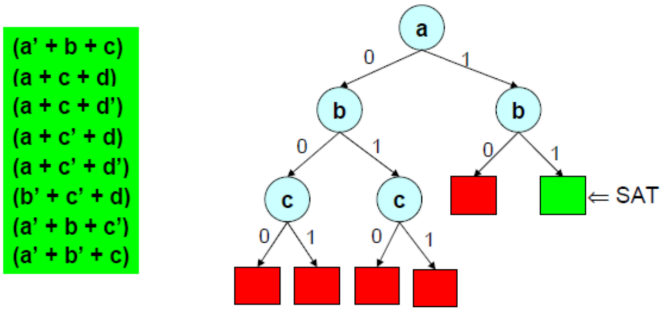

Fig. 2. Basic DPLL Algorithm Pseudo-Code [**?**]

Many other algorithms for addressing the satisfiability problem exist, including conflict-driven clause learning (CDCL) and stochastic local search (eg. WalkSAT). This algorithm was chosen to address the satisfiability problem due to its popularity, performance, and stability. There exist many articles addressing the implementation of DPLL, with generally simple-to-follow steps that allow for easy optimization. Speaking of optimization, different heuristics and tracking methods exist that can help to improve upon DPLL, including MOM, Jeroslow-Wang, Two-literal watching, and so on. DPLL can also be optimized through the use of threading, which allows

for multiprocessing of steps like BCP. For this project, our team set out to address the problem of satisfiability using the DPLL algorithm, making key optimizations to it to improve its speed for larger datasets. Through this dataset, our team hopes to provide a simple and effective method by which to address boolean satisfiability.

March 5, 2023

II. Methods

In order to implement the DPLL algorithm, we began by creating our satSolver.py file, which contains the base DPLL implementation. The first step was to read our benchmark files and parse the provided CNF. This is done in multiple steps using the getlinescleaned(), ispreamble(), iscomment(), and generatecnf(), which parse the lines of the cnf file and output a 2D array containing the literals for each clause. With the CNF formula parsed, we can execute the DPLL algorithm. For our implementation of DPLL we begin with performing unit propagation using the unitpropagation() function. Unit propagation is a major component of DPLL, and works by identifying the unit clauses for a particular variable and removing the negation of that variable from the clauses in the CNF. For our purposes, we chose to have the unit propagation function work in tandem with a separate bcp function, with the unit propagation function executing a while loop to update the CNF using bcp and assign values to the unit clauses identified until no more unit clauses exist. Our team chose to implement a unit propagation function to assist in avoiding unnecessary searches by implementing the while loop seen in the original DPLL in such a way that propagates the unit clauses. Since most time is spent in attempting to execute unit propagation, particularly concerning BCP, our team worked to parallelize this section of the code, which will be discussed below. The next step of the algorithm is to check if the modified CNF is satisfied or unsatisfied. This is done using the clausesallone() and clauseunsat() functions, which check if the current CNF is satisfied or unsatisfied. This is highly important during the recursive backtracking that is built into DPLL, as the function will return to its call whether the variable that was split on is the correct assignment. After this comparison, a new variable is chosen using the Jeroslow-Wang method, which will also be discussed below. Using this variable, DPLL is called recursively with BCP being used to simplify the CNF formula for the positive version of the chosen variable. If this solution is unsatisfied, the negated version of the variable will be attempted during backtracking. Using these methods, we have a simple and efficient implementation for DPLL, which we can further optimize as discussed below.

The Jeroslow-Wang method is a heuristic for selecting a variable in boolean SAT problems. In the standard Jeroslow-Wang (JW) method we assign weights to each variable and the variable with the highest weight is chosen for branching in the SAT solver. The weights are calculated by the length of the clause they show up in with longer clauses being given more weight. In our DPLL algorithm, we use the JW 2-sided method which is a variation of the standard JW method where we find the variable with the largest sum of both the positive and complemented forms weights. The 2-sided method provides better variable selection due to taking into account the polarity of the literals which the standard method does not. Based on the Marques-Silva paper the more common algorithms like BOHM and MOM end up yielding worse results compared to other heuristics. Their interpretation is that these heuristics are "too greedy" and can lead to multiple bad branches. A table provided by the Marques-Silva paper tallying the number of backtracks per heuristic for varying CNFs is provided in Figure 2.

Class	BOHM	DLCS	DLIS	JW-OS	JW-TS	MOM	RAND	RDLIS
aim-200	830	1901	1996	1228	1183	744	1844	1934
bf	1178	404	422	641	725	1219	6360	472
dubois	902	3917	3935	5473	4851	902	15994	4267
ii16	48839	27320	13838	20799	24895	38674	1099	4563
ii32	25892	13868	59	11265	10848	22161	26464	8439
jnh	969	1919	2502	1416	1121	951	9552	2959
pret	1684	1870	1765	1847	1825	1688	43398	1684
ssa	10203	986	739	1139	1410	10202	1159	662
ucsc-bf	40598	12509	9393	11536	12364	38737	31872	14307
ucsc-ssa	61179	5273	4026	6822	16231	61170	15827	5087

Fig. 3. Backtrack Count for Various Heuristics

JW 2-sided performs fewer backtracks for most CNFs compared to both BOHM and MOM, MOM we covered in lecture, and is comparable to a few other heuristics in the graph. RAND was out of the question as it had the "highest percentage of non-chronological backtrack steps" which makes sense as picking randomly can often lead to a bad branch. I will note that these results are gained from a different search algorithm than we are using. Marques-Silva uses the GRASP SAT solver implementation whereas we are using DPLL.

Both the Jeroslow-Wang method and the BCP algorithm were able to be parallelized using the multiprocessing module available in Python3. The parallelization was done by dividing the CNF formula into approximately equal chunks each containing a subset of the clauses. We created multiple processes, each responsible for their portion of the clauses. For JW each process calculated the weights of the variables available in those clauses. For BCP each process assigned the selected variable ran the unit rule on those clauses. Each process independently operates and after they have all completed they retrieve results from a queue and combine them. The multiprocessing module offers many easy threading options due to how the functions take care of most of the memory issues without specific user input. The multiprocessing module was chosen over the threading module due to how the threading module gets bottle-necked by Pythons Global Interpreter Lock. The GIL is a mutex that ensures that only one thread executes Python code at a time. While this makes memory management considerably simpler it effectively makes Python's execution model single-threaded and severely limits the performance of multi-threaded Python programs as the threads cannot fully utilize multiple CPU cores. The multiprocessing module gets around this by using processes instead of threads. Threads share the same memory space and therefore contend for the GIL where processors have separate memory space and each run their own Python interpreter. This allows multiple processes to execute Python code without being affected by the GIL. This does however come with its drawbacks as with multiple processes there is increased memory usage and less efficient resource sharing, however, this second drawback is

mitigated in our program as we do not talk between processes.

III. Conclusion

To talk about the strengths and weaknesses of our SAT solver overall first we need to talk about the specific algorithms implemented. The Jeroslow Wang 2-sided method is a powerful heuristic that considers both polarities of variables and more often than not leads to a decent branch. However, in some SAT instances, the increased complexity as compared to the 1-sided JW method is not worth it as the impact of considering both polarities is minimal. Hence this extra computing power would simply be a detriment in certain cases. The overall algorithm used for our SAT solver is DPLL. DPLL is a complete algorithm which means that no matter what if a solution exists DPLL will find it. It is a very rigorous search procedure that also ensures the correctness of the solution found. With the correct implementation of DPLL, ie. our implementation, there is no question about if the solution found is actually a satisfying arrangement of variables. However, DPLL has an exponential time complexity making it impractical for dealing with massive variable and clause numbers. In such cases, the search space grows exponentially leading to poor computation times. DPLL is also very inefficient when it comes to SAT instances with a high clause-to-variable ratio. In this case, it could spend an exorbitant amount of time on bad branches. Also due to the data structures used to hold the data in the middle of computation, there is a higher memory usage as compared to other SAT solvers. With our parallel implementation, there is a distinct speed decrease with smaller SAT instances, the complexity of our parallel approach leads to a slower output with smaller instances as compared to single-threaded/processed SAT solvers. So our approach is tailored for larger SAT instances with a huge amount of clauses and variables where we can see some speed increase with the multiple processes.

When we started this project we decided on an indexed approach for uniqueness where the solver would take in a sparse array with 0 denoting no variable, 1 denoting a positive variable, and 2 denoting a complemented variable. However, we were not able to get any further than computing small SAT instances with the indexed approach. Based on the time constraint of the project we decided to transfer over to a value-based approach using the input SAT instance file more or less as is. Given more time we would have stuck with this index-based approach and expanded on the matrix manipulation of the sparse matrix representing the CNF. This index-based approach could have been computationally simpler as manipulating that sparse matrix with the matplotlib Python library would have been quick and efficient.

IV. Results

When it comes to larger data sets, the parallel implementation shows its efficiency. With shorter data sets, our algorithm tends to be much slower, as the process of creating workers for each parallel process, as well as waiting for all of the threads to finish calculating their chunk of data and then add or replace that data in the main thread takes a considerable amount of resources. In the example given to us in the project document, it takes around 0.125 seconds to finish, while the single thread takes less than a hundredth of the time to finish. In one of the test files that contains 100 variables and over 400 minterms, it takes less than 6 minutes to compute. We tested our algorithm on Drexel's Xunil server, which is ssh-able Linux server. This was done so that the results could be replicated on similar hardware, considering CPUs and threads are not built the same. Our results can be seen in Figures 4, 5, and 6.

V. Appendix

Fig. 4. SAT instance uf50-0997.cnf

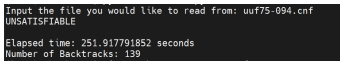

Fig. 5. SAT instance uuf75-094.cnf

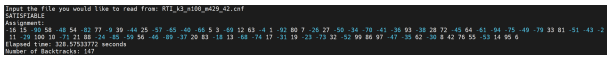

Fig. 6. SAT instance RTI-k3-n100-m429-42.cnf

References

[1] J. Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, pages 62-74, September 1999.