Parallel Strategies for SAT Solver Optimization: A Comparative Study

Yi-Ju Huang (yijuh), Raashi Mohan (raashim)
https://github.com/ivyaccount/pcap_project

Summary

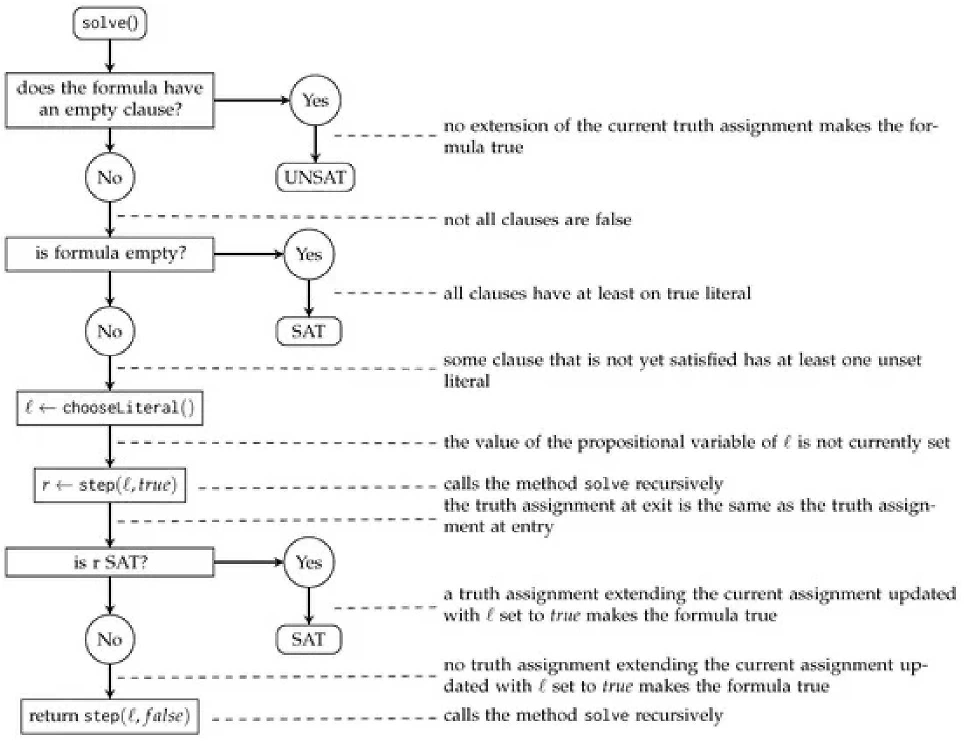
We aim to compare various SAT (Satisfiability) solver algorithms and determine their parallel implementation potential. By leveraging different parallel strategies, such as OpenMP and OpenMPI, on CPUs (specifically, the GHC machines), we intend to analyze and contrast the efficiency of parallelizing different SAT solver strategies and see which strategies can take the most advantage of parallelism.

Background

DPLL (Davis-Putnam-Logemann-Loveland) and CDCL (Conflict-Driven Clause Learning) are both algorithms used for solving the satisfiability (SAT) problem, specifically dealing with Conjunctive Normal Form (CNF) formulas. While they share similarities in terms of their foundational logic for solving SAT problems, they have differences in their approaches and implementation. In our implementations, both algorithms take in a CNF file as input, and output the variables that can be set to true, if the overall clauses have a satisfiable assignment.

DPLL algorithm

The first SAT Solver is DPLL, which utilizes the backtracking technique to fasten the solving process. The flow of the solver can be described as above. If we cannot find out whether the clause set is satisfiable or not, there are three main steps to move on. The first step is unit propagation, we check if there’s any clause that has only one literal. If so, we set the literal value according to its sign and remove the clause from the set if the sign matches, or else only the literal will be removed from the clause. The second step is pure literal elimination, we check if there’s any literal that appears to have the same sign across the set. We remove clauses that contain that literal. Last, we have to guess the value of the remaining literals, which is the branch guessing step. If the guess leads to an unsatisfactory result, we take the opposite sign, otherwise we solve the set by getting a satisfying result.


solve()
does the formula have an empty clause?
Yes
UNSAT
no extension of the current truth assignment makes the formula true
No
not all clauses are false
is formula empty?
Yes
SAT
all clauses have at least on true literal
No
some clause that is not yet satisfied has at least one unset literal
ℓ ← chooseLiteral()
the value of the propositional variable of ℓ is not currently set
r ← step(ℓ, true)
calls the method solve recursively
the truth assignment at exit is the same as the truth assignment at entry
is r SAT?
Yes
SAT
a truth assignment extending the current assignment updated with ℓ set to true makes the formula true
No
no truth assignment extending the current assignment updated with ℓ set to true makes the formula true
return step(ℓ, false)
calls the method solve recursively

The referring flow of algorithm. **[1]**

The base code referring to is using linked lists as its data structure, the clauses are linked together while the literals within each clause are also linked.
The insertion of the linked list only happens when reading the CNF file to transform the numbers into clauses and literals, which is the time that we set up the linked list. The deletion operation can happen during unit propagation or pure literal elimination. The former step removes the literal or clause that appears to be the standalone literal in one of the clauses, the latter one removes the clauses that contain the literal that has the same sign in the entire clause set. The additional operation is deep cloning the entire clause set data structure happening when we cannot downsize the clause set further through unit propagation and pure literal elimination. In this case, we clone the list and guess the value of a literal.

Since the implementation uses an inherently serial data structure, the dependencies happen within every operation about the clause set and the result of back-to-back literal/clause removal. The expensive cost is paid when the literal guess is wrong. We have to backtrack all the way up on the stack to fix that guess.

CDCL Algorithm

The CDCL algorithm is a more efficient extension of the DPLL algorithm mentioned above. CDCL is widely employed in modern SAT solvers due to its ability to handle large CNF formulas effectively. CDCL operates similarly to DPLL but introduces several enhancements, notably conflict analysis and clause learning, to improve its performance.

CDCL begins by assigning truth values to variables and performing unit propagation, similar to DPLL. It explores the search space by making decisions and simplifying the CNF formula using unit propagation and pure literal elimination. This initial phase involves making decisions based on assigned truth values and iteratively refining the formula by eliminating pure literals and performing unit propagation to derive further assignments.
.
When a conflict occurs (a clause becomes unsatisfiable under the current assignment), CDCL identifies the conflict and backtracks to a previous decision level. Unlike DPLL, CDCL employs a more sophisticated mechanism for detecting conflicts, often using efficient data structures like watched literals, which help in quickly identifying conflicts without exhaustive checking. This algorithm then analyzes the conflicts to extract valuable information. It identifies the reason for the conflict by analyzing the conflicting clauses and the assignments leading to the conflict. This process, called conflict analysis, involves learning a new clause based on the conflict, known as a "learned clause" or "assertion clause." These learned clauses help prevent revisiting the same conflicts in future iterations and guide the search towards a solution.

CDCL then incorporates these learned clauses into its search space, effectively learning from encountered conflicts and preventing redundant exploration of the same paths. When backtracking occurs, learned clauses guide the backjumping process, skipping unnecessary search paths that would lead to conflicts. Additionally, a non-chronological backtracking mechanism is employed, allowing for backtracking to a more “advantageous” level in the search tree based on the learned clauses, rather than strictly following the chronological order of decision levels as in traditional DPLL.

Approaches

All of the data is running on the GHC machines with Intel Core i7-9700 processor (3.0 GHz, Eight cores, 8 threads). All OpenMP implementations use the default number of threads.

Parallelizing DPLL With OpenMP

There are two two parts being parallelized. One of them is the file I/O while reading the CNF file to generate the clauses, considering the fact that the order of the clauses does not matter.
The other one is doing the branching (variable value guessing) parallelly. Instead of guessing the value of a variable and backtracking to revise the value, we run both branches at the same time to reduce the time cost.

Since the original code [2] is only in a serial setting, the standard functions and variables they use do not consider if they are thread-safe or not.There are a lot of thoughts going through to make the parallel version work. We have to add critical regions to ensure the access to resources that are shared is mutually exclusive. For example, the file descriptor is a necessary shared variable, therefore the action of reading a file is protected.In contrast, we need to do additional deep cloning of the array of variable's temporary value that is necessary to determine if the clause set is satisfiable or not without tampering other intermediary results. Every standard function being used needs a second thought as well. We have run into a problem of strok() giving indeterminate results, which is caused by using static variables as its internal implementation. There are several ways we were trying. The original thought is parallelizing the linked list operations as it covers the majority of the code (which is one of the reasons why we chose a linked list implementation of DPLL thinking it might be interesting to parallelize this data structure's SAT solver). However, it does not give us a promising result. Understandably, linked lists have a high dependency on the pointer points to each node, it cannot get a satisfying speedup without other structures' help, which complicates the algorithm more.
After looking deeper into the code, we found out the similarity between the fibonacci program and the branching part of the algorithm that gives us the inspiration of variable true/false guess parallelization.

After getting some results with variable false/true guess parallelization, we take the step further trying to adjust the shared variables and parameters of OpenMP. The one we found a speedup relation is setting the threshold whether to generate a task or run serially when the work left is small enough. We decided this by considering the number of remaining variables left that have yet to have a true/false value compared to the original size of the problem (total variable numbers).

Parallelizing CDCL With OpenMP

As a starting point, a C++ implementation of the CDCL algorithm was used [3]. An initial parallelization strategy primarily involved introducing parallelism in picking the branching variable, assigning it, and detecting conflicts across clauses. Within the primary loop of the function, parallelism is introduced in two main phases. In the unit propagation phase, all threads simultaneously aim to propagate unit clauses and derive new variable assignments. Multiple threads make calls to a `unit_propagate` helper function to identify and resolve conflicts. In the second phase, threads collaborate to check if a conflict has occurred — if a conflict is detected, the threads then coordinate to perform conflict analysis and backtrack is necessary. Here, only one thread executes the code to update the `decision_level` and backtrack the assignments to maintain correctness. In this implementation, synchronization mechanisms, such as barriers and atomic operations, were used to control shared variables and ensure that only one thread handles critical tasks.

Unfortunately, the above implementation seemed to perform poorly, failing to complete small test cases after several minutes. One source of this issue could stem from nested parallel regions. Nested parallelism can lead to performance issues or oversubscription if the number of threads created becomes excessive. Additionally, the regular use of `#pragma omp barrier` to maintain thread parity, might introduce synchronization overhead if it's not necessary for all threads to wait at that point.

Moving past this, some success was found in implementing parallelism in the helper functions, rather than in the main function loop. In a modified `unit_propagate_parallel` function, OpenMP directives are used to parallelize the loop that iterates through the clauses to perform unit propagation. Specifically, the function parallelizes the iteration over the clauses, distributing processing of different clauses across multiple threads. Within the parallel loop, identifying unit clauses or unsatisfied clauses are executed atomically, preventing race conditions when modifying shared variables. Ultimately, this approach splits the clause iteration work among threads, allowing concurrent processing, while critical sections safeguard shared variables against simultaneous modifications, guaranteeing data consistency and preventing conflicts among the threads.

Additionally, the SAT solver employs OpenMP directives to parallelize the selection of a branching variable. Multiple threads are engaged in searching for an unassigned variable randomly within a specific range — this simultaneous effort allows for multiple threads to attempt variable selection in parallel, aiming to reduce the overall computation time by concurrently exploring different possibilities for the branching variable selection. However, the separation of a critical section ensures that multiple threads trying to select an unassigned variable randomly do so safely, preventing simultaneous access and modification of shared resources.

Overall, the structure of the original sequential implementation was preserved. Updates for parallelism were made by adding OpenMP directives to sections of helper functions, and modifying these functions slightly to preserve critical sections.

Parallelizing CDCL with OpenMPI: Using HordeSAT Portfolio

To move past the implementation with OpenMP, we looked to implement parallelism methods via message passing with OpenMPI. In researching methods to construct this approach in the time left, we found "HordeSAT", a parallel SAT solver framework designed for multi-core and distributed-memory systems intended to efficiently solve SAT problems by employing parallel computing techniques [5]. HordeSAT implements various parallel algorithms and strategies to exploit the computational power of modern parallel architectures, such as multi-core CPUs and distributed systems. Its goal is to enhance the performance of SAT solving by distributing the workload across multiple cores or nodes, thereby aiming to solve SAT instances more quickly.

HordeSat employs a portfolio approach that involves running multiple instances of SAT solvers concurrently on the same problem until one solver finds a solution. This portfolio can consist of instances of a single solver with different configurations or entirely different solvers. The solvers in the portfolio communicate by exchanging learned clauses, allowing them to avoid redundant work and potentially improve performance. The idea is to diversify the search space exploration by using various solver settings and exchanging useful information among solvers. The solver utilizes a decentralized design (there is no central node managing the search or communication). Instead, all nodes are equivalent and work independently. It employs hierarchical parallelism, taking advantage of both shared-memory parallelism within nodes and message passing between nodes in a cluster.

In terms of parallelizing the CDCL algorithm, HordeSat distributes the workload among multiple cores. Each core handles a different instance of a SAT solver, known as a "core solver," operating independently to explore the search space. Communication between core solvers occurs through OpenMPI, allowing them to exchange learned clauses and avoid redundant exploration of the search space.

Ultimately, HordeSat achieves load balancing in its massively parallel environment through a decentralized design, dynamically allocating multiple SAT solvers across processing nodes. The decentralized architecture allows each solver to operate independently, adapting its workload without centralized control. It employs hierarchical parallelism, utilizing shared-memory within nodes and message passing between nodes, and diversification strategies to evenly distribute the search space exploration. Periodic clause exchange between solvers reduces redundant exploration and balances workload by sharing learned information. These combined measures aim to optimize resource utilization, prevent idle time, and ensure a more uniform distribution of computational tasks among solvers and nodes.

Given a header file that specified the names of sub-functions that needed to be included, we were able to recreate the sequential CDCL code used in the earlier part to fit the specifications for the HordeSAT portfolio, in C++. Minor changes were made to the main portfolio source code, so that it could support our CDCL version, but for the most part it was left intact [5].

Results

Experiment Setup

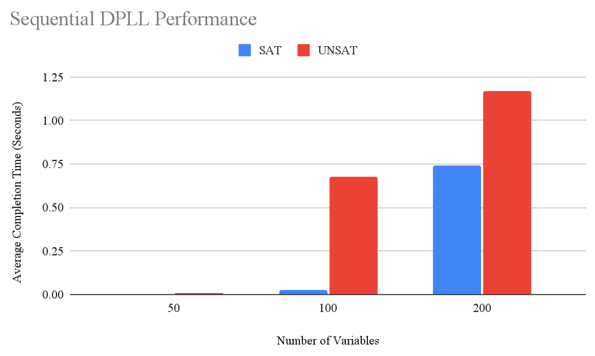
Each experiment is run on GHC machines with Intel Core i7-9700 processor (3.0 GHz, Eight cores, 8 threads). There are 72 test cases in total, 24 tests with 50 variables, 100 variables and 200 variables each. 47 of the test cases are Satisfiable and 25 are not. The test cases were selected from the SATLIB Solvers Collection **[4]**.

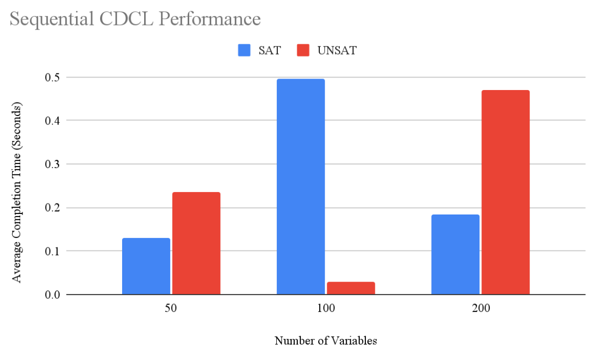
In terms of machine choice, using a GPU would have required a different implementation. Additionally, many existing projects that concern SAT solvers focus on adapting algorithms to GPU use, rather than determining sources of CPU parallelism. Additionally, for reference, each OpenMP implementation used the default number of threads on the system.

Measuring Performance

We are using Linux's perf tool, with more detailed statistics, to help us measure the time it takes to generate the solution. The cache data is also measured with the same tool. We automated the process with scripts to run all test cases, aggregate necessary data from the terminal output, and populate a csv file with the results.

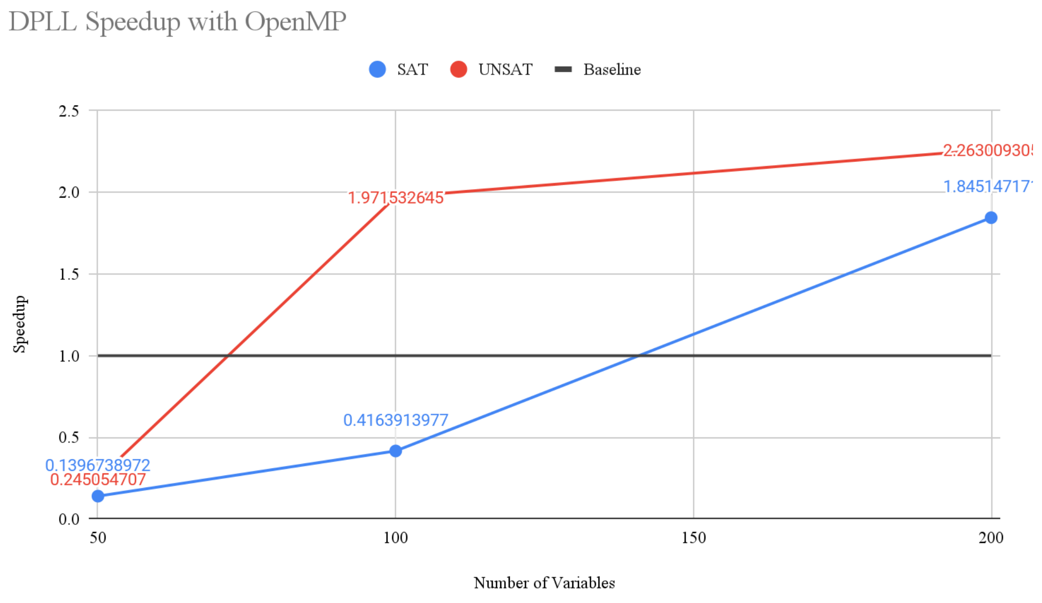
Preliminary Analysis





The above bar graphs show the average completion times for the two algorithms - the test suite included tests with 50, 100, and 200 variables, with both Satisfiable and Unsatisfiable problems. (note here that the scale of average computation time is different in both charts). The DPLL algorithm seems to be slower than the CDCL algorithm on the 200 variable cases, but overall both algorithms seem to perform relatively quickly.

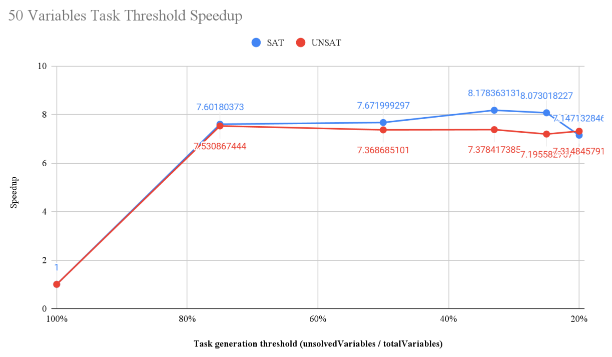
DPLL Algorithm - OpenMP

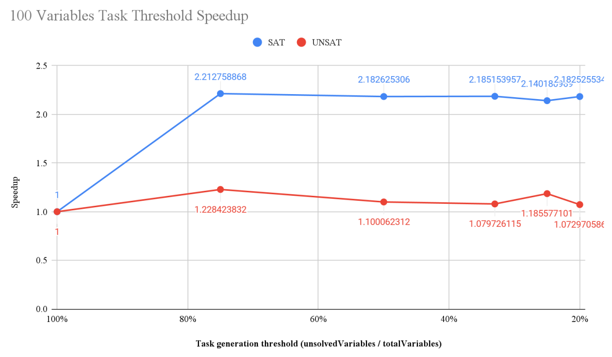


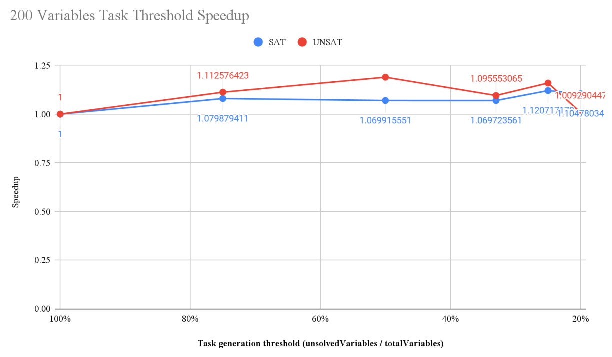
Final speedup

The overall speedup of our parallel DPLL SAT solver result is shown above. The baseline is our sequential version of DPLL, and we are computing the speedup by dividing the baseline completion time with our parallel version's complete time. We can see the trend that when there's a high demand for computation, we can achieve a better speedup. It is either when there's a lot of variables to go through or when we have to run through more of the possibilities (unsatisfactory case).

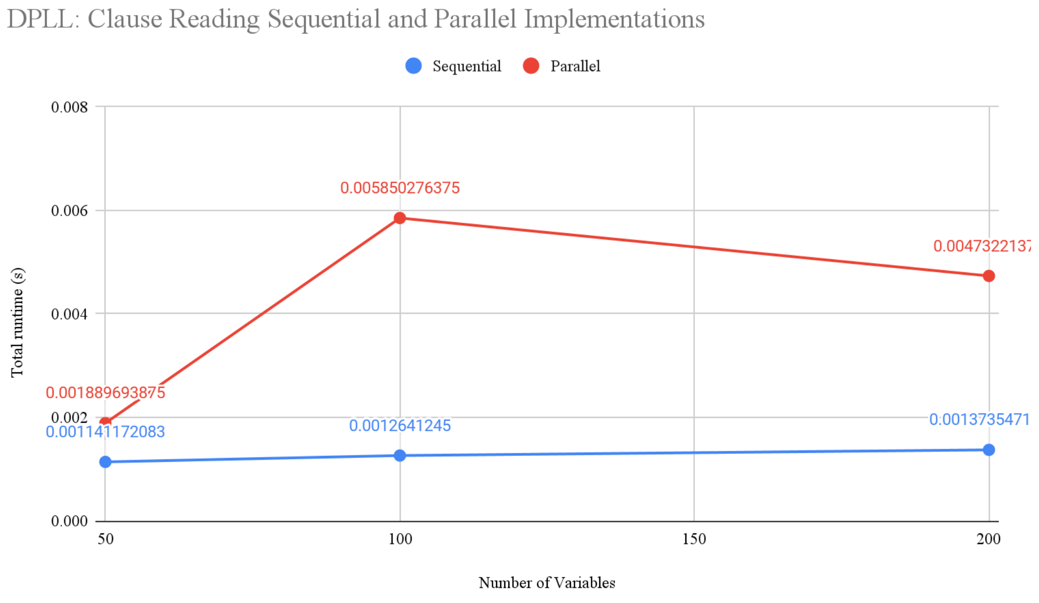
The speedup comparison of our parallel DPLL SAT solver with different task running results are shown below. The baseline completion time is when we did not add any threshold for the OpenMP tasks. It means that no matter how many variables are left to compute, we will run all the cases with a task when reaching the true/false guessing stage. Otherwise, the percentage means the ratio of the variables remain to be decided versus the total variables in the clause set. 33 percent means that when one third of the variables' values are undetermined, we can run the code without branching tasks to complete. All the threshold we set guarantees us to gain speedup compared to the ones without. Taking the average of SAT and UNSAT cases, the best speedup we can get is around the range of 25-33% of the total variables as threshold.



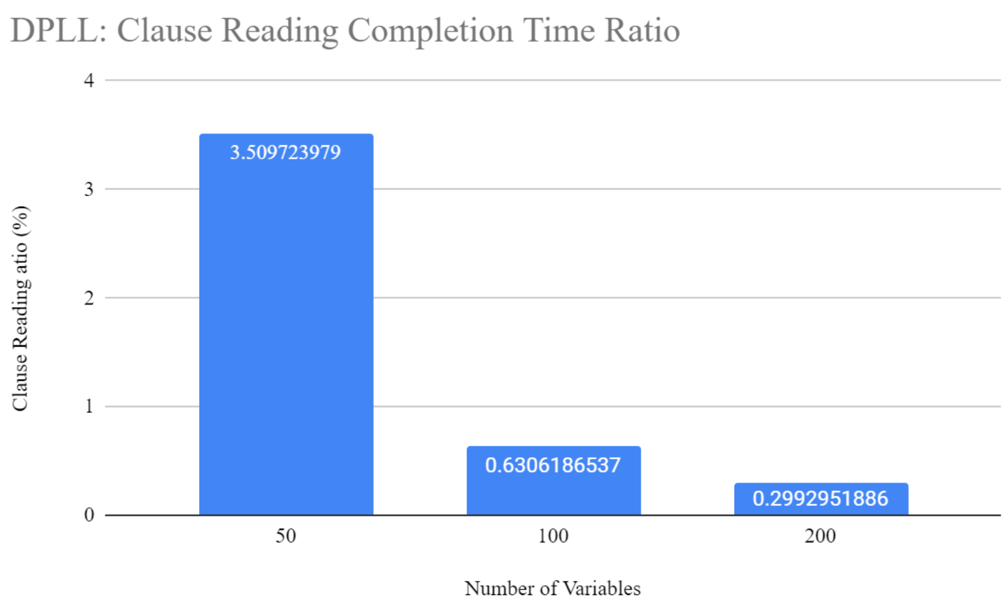




Speedup of setting different task running threshold



As for the result of parallelizing the reading clause section, we did not get an ideal speedup as expected. Shown in the graph above, it shows that paralleling the process even adds additional overhead. After digging through the data and code, there are a mix of problems causing this. Since it is a file I/O operation, most of the time is spent in this critical section reading the content out. We also saw that the task is not evenly distributed, some threads take on more clause generating (relatively heavy loaded), while some others deal with the comments (light weighted) that barely needs computation.



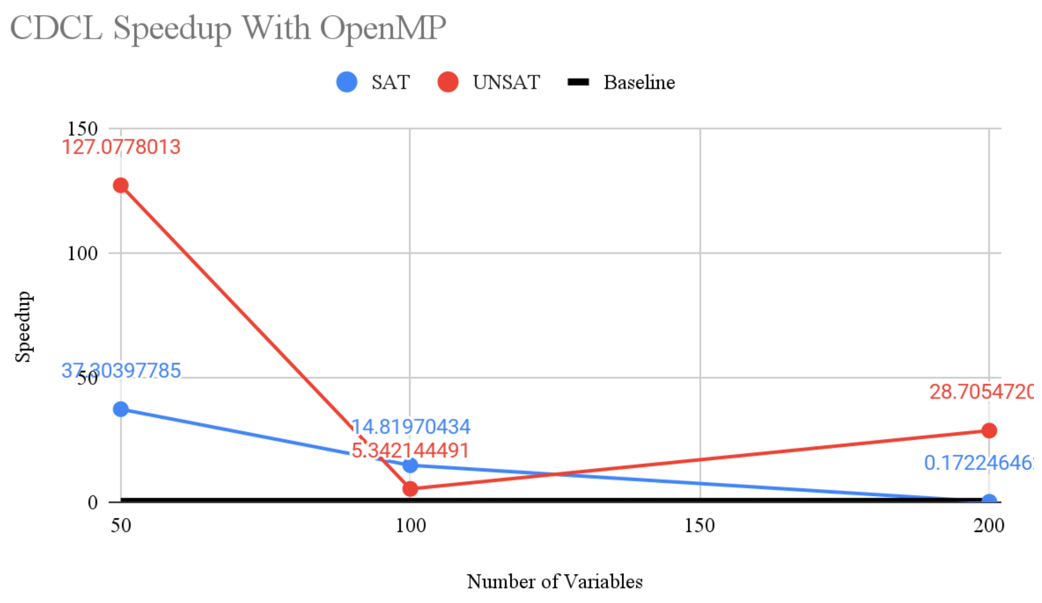
However, the time spent in reading the clauses only takes a small percentage of the total completion time, most of the time is spent on running the DPLL algorithm (the graph above shows the actual percentage. Considering this fact, the main time is definitely spent on the expensive deep cloning operations and the need to traverse the list again and again during unit propagation and pure literal elimination. With that, no matter what machine target we choose, we will end up struggling with the inherently serial part of the data structure. Possible next step is we can keep the list structure but a better traversing support with forward list[1] in C++, that provides erase operations anywhere within the sequence. With the migration to this data structure we may reduce the traversal cost during unit propagation and pure literal elimination.

CDCL Algorithm - OpenMP

Below, we have the computational speedup with the CDCL algorithm implementation parallelized with OpenMP. We can see that for most of the cases, the speedup is overwhelmingly positive (with a maximum speedup of 127x). The positive speedup achieved with 50 variables indicates that parallelizing the CDCL algorithm with OpenMP yielded substantial benefits. As the number of variables increased to 100 and 200, the speedup declined, suggesting that the

[1] https://cplusplus.com/reference/forward_list/forward_list/

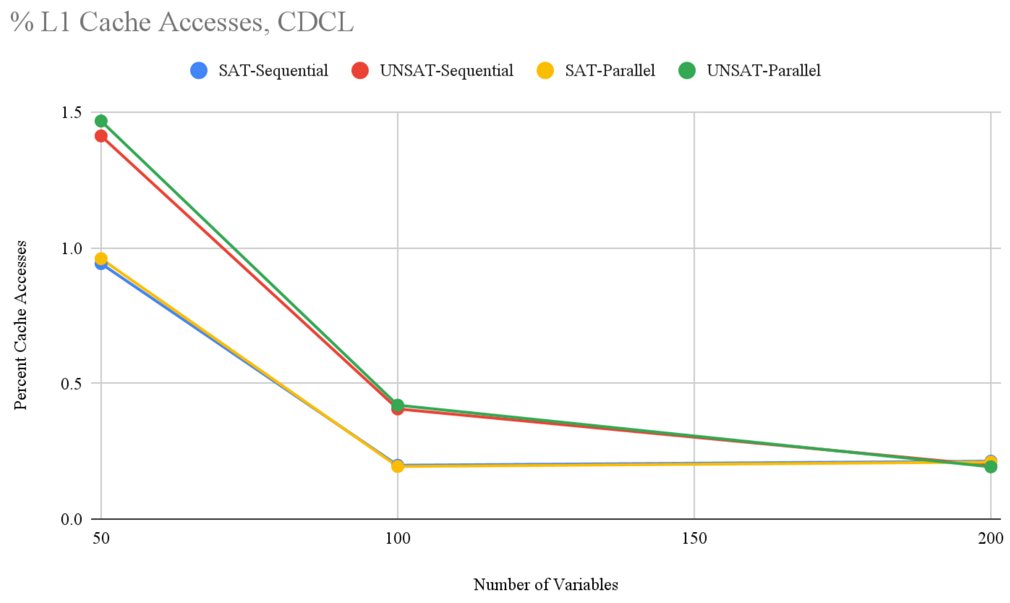
parallelization strategy might encounter limitations or inefficiencies when handling larger problem instances.



Several factors could contribute to this observed decrease in speedup. One potential factor could be increased contention for shared resources or increased overhead related to managing parallel threads as the problem size grows. When the number of variables increases, the workload might not be distributed as efficiently among the threads, leading to less effective parallelization. Furthermore, the CDCL algorithm's inherent characteristics might affect the scalability of parallel execution. As the problem size expands, the algorithm's workload distribution, memory access patterns, or inherent parallelism might pose challenges for achieving continued speedup.

Another trend to note here is that in many cases, the speedup observed for unsatisfiable cases tends to be faster compared to satisfiable cases. We believe that this is because the SAT solver may identify inconsistencies or conflicts earlier in the search process. When a conflict is detected, the solver often performs a backtracking mechanism to resolve the issue and refine the search space, which can lead to earlier termination. As a result, the workload might be reduced, allowing the parallel threads to converge more quickly towards the contradiction, potentially benefiting from a higher speedup.

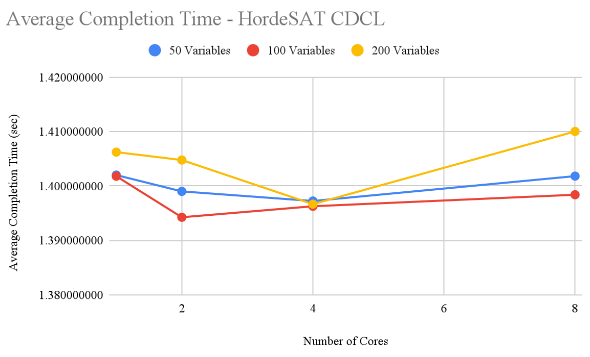
The metric "% of all L1 accesses" in performance analysis usually indicates the proportion of accesses to the Level 1 data cache among all memory accesses. The average results of this metric are shown below. When comparing sequential and parallel implementations, the similar percentages of L1-dcache accesses we can see here imply that both execution modes exhibit analogous memory access patterns.

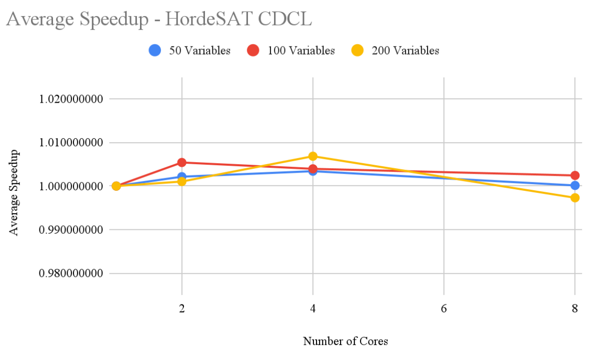


Additionally, convergence of this percentage at 200 variables, both for satisfiable and unsatisfiable cases in sequential and parallel implementations, implies that the memory access behavior stabilizes or becomes consistent for larger problem sizes. This stability might indicate that the workload or memory access patterns reach a certain equilibrium, regardless of the parallelization strategy or problem satisfiability.

Lastly, we can see that the percentage of L1-dcache accesses is higher for unsatisfiable tests compared to satisfiable cases. This could be due to the fact that unsatisfiable instances often involve more complex computations or a larger number of operations than satisfiable ones. These more complex computations might result in increased memory accesses, leading to a higher percentage of L1-dcache accesses.

CDCL - HordeSAT Portfolio





Above we can see the performance results of the test suite on the HordeSAT portfolio, with 1, 2, 4, and 8 cores. Here, we can see that the completion time decreases and speedup increases when moving from 1 to 2 cores, and then again a bit when moving from 2 to 4 cores, but this trend isn't seen from 4 to 8 cores. It is likely here that there is the emergence of increased communication overhead as the number of cores exceeds a certain threshold (in this case, 4 cores). As more cores are added, the communication between them becomes more frequent and data exchange overhead becomes more pronounced. This additional communication overhead might negate the advantages gained from parallelism, leading to diminishing returns or even decreased efficiency in terms of completion time and speedup. This behavior suggests that the communication cost between cores in the HordeSAT portfolio solver becomes more significant as the number of cores increases beyond a certain point. Consequently, the benefits of parallelism become less pronounced, resulting in a plateau or decline in performance improvement.

References

[1] Andrici, C. C., & Ciobâcă, Ș. (2022). A Verified Implementation of the DPLL Algorithm in Dafny. Mathematics, 10(13), 2264.
[2] https://github.com/uzum/dpll-sat-solver/tree/master
[3] https://github.com/sukrutrao/SAT-Solver-CDCL
[4] https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/DIMACS/AIM/descr.html
[5] https://doi.org/10.48550/arXiv.1505.03340

Distribution of Work

The distribution of work was 50%-50%. Yi-Ju worked on the DPLL algorithm, while Raashi worked on the CDCL algorithm.

Yi-Ju

- OpenMP parallelism of DPLL algorithm
- Identifying initial sequential DPLL and CDCL implementations
- Compiling performance results of DPLL algorithm

Raashi

- OpenMP parallelism of CDCL algorithm
- HordeSAT portfolio implementation of CDCL algorithm
- Create test scripts and compiling performance results of CDCL algorithm