ME/CS/ECE759 Fall 2023
Final Project Report

TeamSAT

A Collaborative Parallel SAT Solver

Fariha Tabassum Islam
Md. Tareq Mahmood

December 13, 2023

Abstract

The SAT problem involves determining the satisfiability of logical formulas, crucial in diverse applications such as artificial intelligence, cryptography, and optimization. The CDCL algorithm, a cornerstone of modern SAT solvers, employs conflict analysis and dynamic clause learning to efficiently navigate the SAT problem. Recognizing the computational complexity of SAT, the study explores parallelization strategies using MPI, PThread, and OpenMP. The MPI approach, involving multi-node concurrent solvers who shares learned clauses, exhibits superior performance, particularly in handling larger problem instances. The impact of parallelism is evident across varying problem sizes, with optimal configurations depending on problem characteristics. The results showcase the scalability and efficiency of parallel SAT solvers, emphasizing their significance in addressing real-world challenges.

Contents

1 Problem Statement

This section delves into the rationale and foundations driving SAT problem-solving. We explore a prominent algorithm employed by modern SAT solvers and scrutinize the structural elements characterizing a general SAT problem. This concise overview aims to provide a scholarly understanding of key aspects, encompassing algorithmic strategies and the inherent structural framework of the SAT computational challenge.

1.1 Boolean Satisfiability Problem

The Boolean Satisfiability Problem (SAT) involves determining if a given logical formula, expressed in propositional logic with Boolean variables, can be satisfied by assigning truth values to the variables. The goal is to find a combination of truth values that makes the entire formula true. Formally, given a set S of m clauses in CNF form and n unknown boolean variables ($v_i \in \{0, 1\}$), the goal is to find an assignment for each variable so that all clauses $c_i \in S$ is satisfied.

Consider the formula $(v_1 \vee v_2) \wedge (\neg v_1 \vee v_3)$. This formula has $m = 2$ clauses and $n = 3$ variables. It queries if there exists a truth assignment $(v_1, v_2, v_3) \in \{0, 1\}$ that makes the entire formula true. For instance, assigning $v_1 = 0, v_2 = 1, v_3 = 1$ satisfies the formula.

SAT is NP-complete, meaning its solution difficulty grows exponentially with the number of variables. Despite its computational complexity, SAT is fundamental in various applications, serving as a foundation for formal verification, artificial intelligence, optimization, and cryptography. Efficient solutions to SAT, such as SAT solvers, have broad implications in problem-solving across diverse fields.

1.2 CDCL Algorithm

The Conflict-Driven Clause Learning (CDCL) algorithm [1] is a fundamental component of modern SAT solvers, first proposed in [1, 2]. It employs a backtracking search augmented by conflict analysis. CDCL dynamically constructs a directed acyclic graph of decisions and learned clauses during search. When a conflict arises, it backtracks to a decision level, learning a clause representing the conflict, and applies a conflict-driven learning mechanism. This learned clause prunes the search space, preventing similar conflicts. CDCL exploits a watchlist mechanism to efficiently propagate assignments. The algorithm efficiently navigates the Boolean satisfiability problem by leveraging learned information, significantly improving solving capabilities for complex instances.

[1]Wikipedia Article: `https://en.wikipedia.org/wiki/Conflict-driven_clause_learning`

1.3 Why Build A (Parallel) SAT Solver?

SAT solvers are crucial in various fields due to their ability to solve Boolean satisfiability problems efficiently. These problems arise in diverse applications such as hardware and software verification, artificial intelligence, cryptography, and optimization. By determining whether a logical formula is satisfiable, SAT solvers aid in decision-making, problem-solving, and automated reasoning. Their efficiency enables the solution of complex problems with numerous variables and constraints, contributing to advancements in technology, algorithm design, and problem-solving methodologies across a spectrum of disciplines.

Parallelizing SAT solvers is imperative for tackling complex problems efficiently. By harnessing the power of multiple processors or cores, parallelization accelerates the exploration of the solution space, significantly reducing solving times for large instances. This is especially critical in domains such as formal verification and artificial intelligence, where intricate problems demand substantial computational resources. Parallel SAT solvers enhance scalability, allowing effective handling of increasingly larger problem sizes. The concurrent processing capability not only improves overall performance but also enables the resolution of real-world challenges with numerous variables and constraints, making parallelization an essential advancement in optimizing SAT-solving capabilities.

1.4 Problem Structure

A `.cnf` file, used to encode Boolean satisfiability (SAT) problems, consists of a header specifying the number of variables and clauses. Each clause is a line of integers representing literals, with a terminating `0`. Positive integers denote variables, while negative ones indicate negations. The sample SAT problem in Section 1.1 would look like this.

```
c simple_v3_c2.cnf
c
p cnf 3 2
1 2 0
-1 3 0
```

A SAT solver will take a `.cnf` file as input and output (i) whether the formula is satisfiable, (ii) if satisfiable, the assignment that makes the formula true.

2 Solution Description

Commencing with a basic and single-threaded implementation of the CDCL algorithm in Python[2], we embark on a systematic transpilation of the codebase to `C++`. This initial **cpp** variant represents a single-node, single-threaded solver. Subsequently, we diverge into two distinct parallelization approaches:

- **cpp_mpi:** This variant involves the development of multi-node concurrent solvers using `OpenMPI`, facilitating the exchange of learned clauses among nodes for enhanced collaborative problem-solving.
- **cpp_omp:** This variant entails the implementation of a single-node, multi-threaded solver utilizing `OpenMP`, thereby exploiting parallelism at the thread level for improved computational efficiency.
- **cpp_pthread**: This variant also constitutes a single-node, multi-threaded solver utilizing the `pthreads` library, offering fine-grained control over thread-level parallelism.

The subsequent section delves into a comprehensive exploration of the methodologies employed in parallelizing the CDCL algorithm, highlighting the intricacies and outcomes of our parallelization efforts.

2.1 MPI Implementation: `cpp_mpi`

The primary intuition behind our methodology is to concurrently run multiple CDCL solvers, each exploring different branches while searching for a satisfiable assignment. Upon encountering a conflict and subsequently learning a new clause, a solver disseminates this knowledge to other parallel solvers. The collaborative nature of our approach aims to exploit the diversity of exploration strategies, enhancing the overall solver performance. The anticipated outcomes of this parallelization strategy are twofold:

1. **Increased Clause Learning:** The collective effort of parallel solvers is expected to yield a more extensive set of learned clauses compared to a single solver. The proliferation of clauses introduces additional dependencies among variables, contributing to more effective branch pruning and, consequently, improved overall solver efficiency.
2. **Diverse Exploration Orders:** Given that finding a solution can be highly contingent on the order of random assignments, running multiple solvers concurrently exploring different orders enhances the likelihood of discovering a (sub-)optimal order. This diversification in exploration strategies proves beneficial in overcoming the inherent sensitivity of CDCL solvers to assignment order.

In our proposed framework, we initialize a system comprising N nodes (where $N \geq 3$),

[2]`https://github.com/apurva91/SAT-Solver`

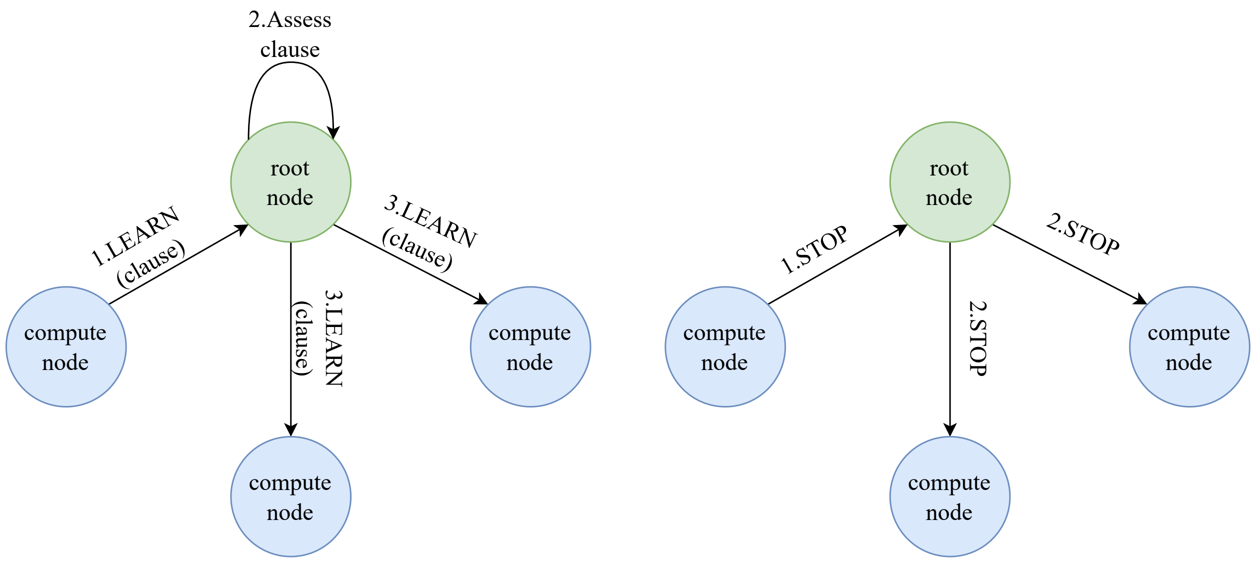

Figure 1: Overview of the MPI approach

with one node designated as the root and the remaining $N-1$ nodes serving as compute nodes. Each compute node is assigned a distinct random seed, ensuring diverse branching strategies. Upon discovering a new clause, a compute node transmits a LEARN message to the root node, which centrally accumulates and manages all acquired learnings from the compute nodes. If the identified clause remains unexplored by other compute nodes, the root node disseminates the new clause to all other nodes, fostering collaborative learning. The termination criterion is met when any compute node successfully identifies a solution, prompting the issuance of a STOP message to the root node. The root node promptly relays the termination signal to all other compute nodes, triggering an immediate halt in their processing. This approach facilitates efficient collaboration and knowledge sharing among compute nodes while ensuring a synchronized conclusion upon achieving a solution.

To distinguish between the LEARN and STOP messages in our communication protocol, we have implemented custom MPI tags. The STOP messages are defined as empty messages containing zero bytes of data.[3] Both the root and compute nodes employ `MPI_Iprobe` in conjunction with `MPI_ANY_TAG` and `MPI_ANY_SOURCE` to assess the presence of incoming messages, ensuring efficient and tag-specific message handling within our parallel computing framework. Given that learned clauses can vary in size, the `MPI_Iprobe` function proves invaluable by providing an `MPI_Status`, which furnishes essential information such as the source and size of the incoming message.

To streamline the efficient dispatch of messages, we initially adopted `MPI_Isend`; however, as we observed a notable frequency of message transmission, especially from the root's standpoint, we identified a potential performance bottleneck. In response, we transitioned to utilizing `MPI_Bsend`, incorporating a sufficiently large attached buffer. This strategic

[3]It is worth noting that the MPICH implementation of MPI imposes restrictions on sending messages with NULL buffers. In response to this limitation, we have transitioned to using OpenMPI.

shift has proven successful in mitigating the performance concerns associated with the high frequency of message sending.

2.2 PThread Implementation: `cpp_pthread`

In this section, we present the details of our `pthread`-based implementation, denoted as `cpp_pthread`. We maintain a structure akin to the high-level design employed in the MPI implementation (Section 2.1). Specifically, the main thread is designated as the *Coordinator*, while each additional thread serves as an independent *SAT solver*. To expedite the process of finding a solution, SAT solvers share their intermediate findings with the coordinator, which, in turn, disseminates this knowledge to all other SAT solvers. The objective was to enhance the overall search speed by leveraging shared insights. All threads have access to a shared variable, `solved`, initially set to `false`. When any thread discovers a solution, it sets `solved` to `true`. All SAT solvers, including the coordinator, routinely check the state of this shared variable, and terminating when a solution is found.

To enhance the efficiency of our implementation, we made key design choices to minimize shared data and reduce critical regions. **Coordinator Side:** The coordinator runs in a loop, dealing with shared data for new clauses. We optimized this by using *conditional variables* and *signals*. These tools allow the coordinator to acquire locks only when there are new clauses to process. We also keep the history of clause previously received, so that coordinator does not propagate a clause twice. This way, SAT solvers do not waste compute time processing the same data. **SAT Solver Side:** Each SAT solver uses a shared buffer for received clauses. Instead of making its entire clauses list shared data, the solver adds received clauses to its list in each iteration. This approach prevents a significant part of the solving algorithm from falling under critical regions, preserving performance.

2.3 OpenMP Implementation: `cpp_omp`

In this section, we present the details of our OpenMP implementation, denoted as `cpp_omp`. Our initial design aimed to mirror the structure employed in the `pthread`-based implementation. Our algorithm doesn't offer many chances for splitting its tasks into different parts and running them simultaneously or data-level parallelism. Instead, it's more fitting for task-level parallelism, where we run multiple copies of the same algorithm concurrently and share their findings. However, we encountered challenges in implementing an effective signaling mechanism between the SAT solvers and the coordinator using OpenMP `pragmas`. We wanted more fine grained control over our shared variables, therefore we decided to use `pthread` instead (Section 2.2).

In our OpenMP version, we utilized some data-level parallelism through `#pragma omp parallel for`, `#pragma omp critical`, and some task-level parallelism through `#pragma` `omp parallel sections`. We sanitized and formatted clauses parallelly. Subsequently, we used a `critical` region to add them to a shared variable. We parallelized two independent tasks, backtracking graph and backtracking history, using `parallel sections` concurrently in each iteration. Despite these efforts, due to limited opportunities for parallelization, this OpenMP version performs less efficiently on larger problems compared to cpp_mpiand cpp_omp. However, it still outperforms the sequential algorithm.

3 Overview of Results. Demonstration of Project

This section addresses the selected test cases employed to benchmark diverse variants of our methodologies. The ensuing discussion will delve into the obtained results, providing insights into the underlying reasons for the observed performance outcomes.

3.1 Test Cases

To facilitate a comprehensive comparison of the solvers, we selected four SAT problems with varying complexities, as detailed in Table 1. These problems span a spectrum of increasing difficulty, enabling a nuanced evaluation of solver performance across different levels of problem intricacy.

Table 1: Overview of the test cases

Test Case	# Variables	# Clauses	Hardness
flat30-13	90	300	Easy
19x19queens	361	10735	Medium
bmc-2	2810	11683	Medium
bmc-7	8710	39774	Hard

3.2 Comparison of Approaches

In Figure 2, a high-level comparison of our methodologies' best-performing variants is presented. Notably, `cpp_mpi` exhibits superior speed in three out of four cases. The efficiency gains of `cpp_mpi` become more pronounced with larger problem instances, emphasizing its scalability and effectiveness in handling increased variables and clauses.

The impact of parallelism becomes evident across varying problem sizes. In the case of smaller problem sizes, such as `flat30-13`, parallelism exhibits minimal effect and, in some instances, may introduce overhead. However, as the problem size moderately increases, single-node parallelism, as observed in `19x19queens`, begins to demonstrate benefits. For

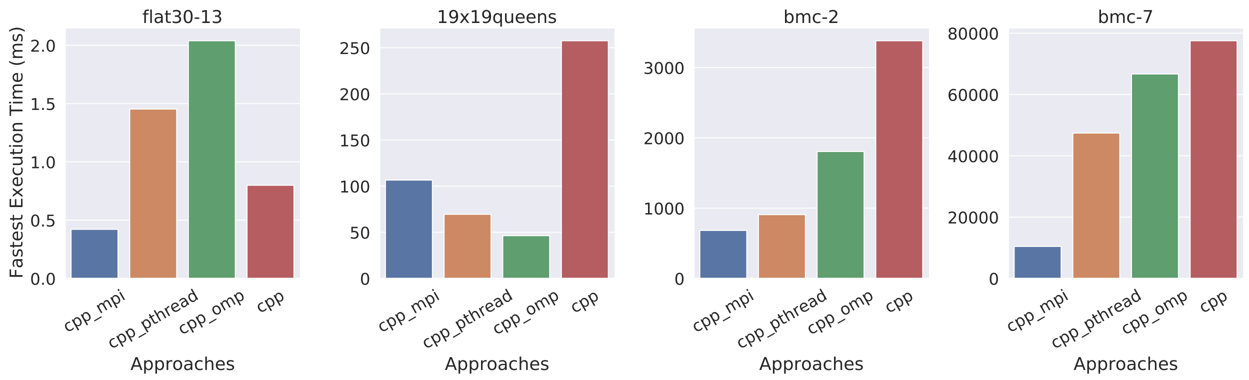

Figure 2: Comparison among the fastest variant of different approaches

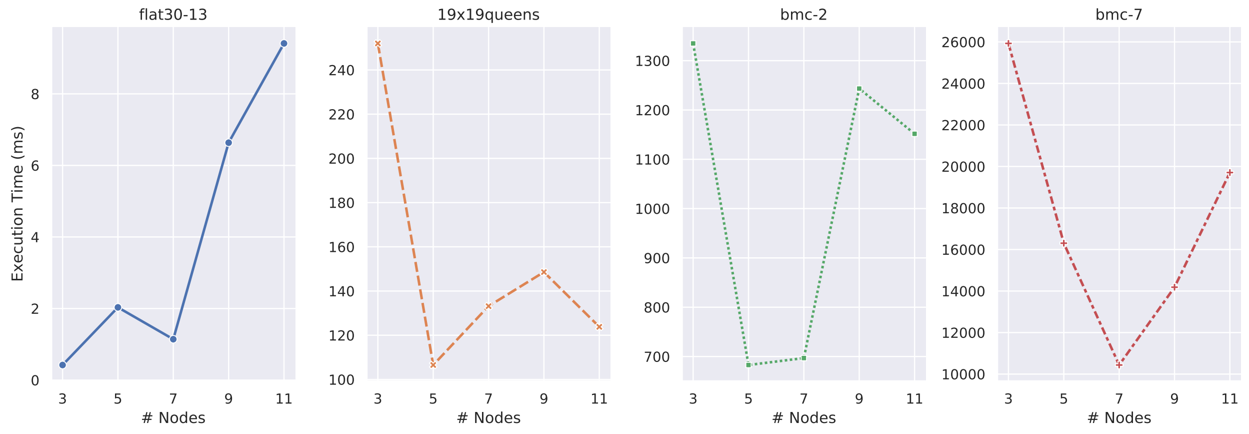

Figure 3: Performance of `cpp_mpi` on varying number of nodes

multi-node concurrent solvers, the degree of improvement depends on inter-connection latency, resulting in modest gains for medium-sized problems. Notably, for larger problem sizes like `bmc-7`, the substantial performance improvement outweighs the inter-connection overhead, showcasing significant enhancements in solver efficiency.

3.3 Variants of MPI Approach

To comprehend the influence of the number of nodes on solution speed, we conducted experiments varying the node count from 3 to 11 on the *Euler* cluster. The results, depicted in various plots in Figure 3, underscore a consistent trend: as the number of nodes increases, parallelism proves beneficial. However, beyond a certain point, additional nodes introduce network overhead, diminishing the performance gains. Intriguingly, the optimal number of nodes exhibits a positive correlation with the problem size.

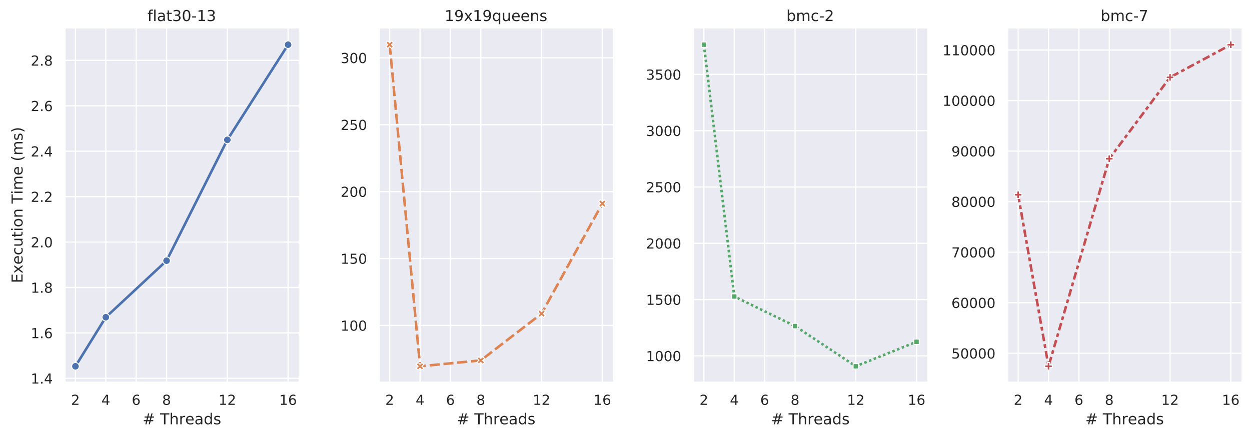

Figure 4: Performance of `cpp_pthread` on varying number of threads

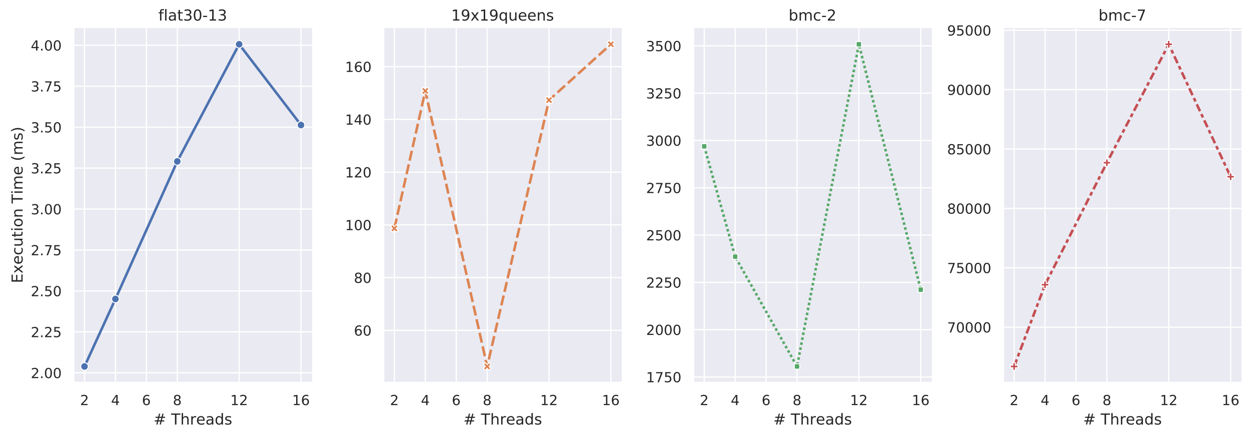

Figure 5: Performance of `cpp_omp` on varying number of threads

3.4 Variants of PThread Approach

A parallelism trend similar to previous observations is evident here too, where the optimal number of threads increases with larger problem sizes. The nature of the SAT problem plays a significant role, with some formulas being inherently easier to solve due to heightened variable dependencies. This insight is exemplified by the efficient performance of `bmc-7` on fewer nodes, highlighting the impact of problem-specific characteristics on the interplay between parallelization effectiveness and SAT instance complexity.

3.5 Variants of OpenMP Approach

Our OpenMP approach has only one solver with some data and task level parallelism within it. These parallelisms often require small number of threads. Hence, large number of threads will only result in overheads. This is evident in Figure 5. Also, our initial guess is aligned that, OpenMP is less suitable for such type of parallelisms.

Our OpenMP approach incorporates a single solver with a combination of data and task-level parallelism. These parallelization strategies we adopted often necessitate a small number of threads. As illustrated in Figure 5, introducing a large number of threads results in overheads rather than performance gains. This observation aligns with our initial hypothesis, suggesting that OpenMP might be less suitable for this specific type of algorithm.

4 Deliverables: Building & Running the Project

We recommend reading our `Readme.md` file inside the `FinalProject` folder. This will facilitate the seamless copying of instructions and execution of the code.
Link: `https://git.doit.wisc.edu/TAREQ/repo759/-/blob/main/FinalProject/Readme.md`

Our datasets are available in the following directory: `FinalProject/test_cases/` There are four files:

1. `FinalProject/test_cases/flat30-13.cnf`
2. `FinalProject/test_cases/19x19queens.cnf`
3. `FinalProject/test_cases/bmc-2.cnf`
4. `FinalProject/test_cases/bmc-7.cnf`

4.1 Baseline

Code: `FinalProject/cpp/sat_solver.cpp`
Script to run: `FinalProject/cpp/run.sh`, `FinalProject/cpp/slurm-build.sh`, `FinalProject/cpp/slurm-run.sh`
Example commands to run in slurm:

- `$ cd FinalProject/cpp/`
- `$ sbatch slurmbuild.sh`
- `$ sbatch slurmrun.sh`
- `$ cat output.txt`

Please edit `slurmrun.sh`, if you want to change the default test case.
Example commands to run in your own machine:

- `$ cd FinalProject/cpp/`
- `$ ./run.sh ../test_cases/flat30-13.cnf`

- `$ ./run.sh ../test_cases/19x19queens.cnf`
- `$ ./run.sh ../test_cases/bmc-2.cnf`
- `$ ./run.sh ../test_cases/bmc-7.cnf`

4.2 `cpp_mpi`

Code: `FinalProject/cpp_mpi/sat_solver.cpp`
Script to run: `FinalProject/cpp_mpi/run.sh`, `FinalProject/cpp_mpi/slurm-build.sh`, `FinalProject/cpp_mpi/slurm-run.sh`
Example commands to run in slurm:

- `$ cd FinalProject/cpp_mpi/`
- Same as baseline

Please edit `slurmrun.sh`, if you want to change the default test case.
Example commands to run in your own machine:

- `$ cd FinalProject/cpp_mpi/`
- Same as baseline

4.3 `cpp_pthread`

Code: `FinalProject/cpp_pthread/sat_solver.cpp`
Script to run: `FinalProject/cpp_pthread/run.sh`, `FinalProject/cpp_pthread/slurm-build.sh`, `FinalProject/cpp_pthread/slurm-run.sh`
Example commands to run in slurm:

- `$ cd FinalProject/cpp_pthread/`
- Same as baseline

Please edit `slurmrun.sh`, if you want to change the default test case.
Example commands to run in your own machine:

- `$ cd FinalProject/cpp_pthread/`
- `$ ./run.sh ../test_cases/flat30-13.cnf` 4
- Similar

4.4 `cpp_omp`

Code: `FinalProject/cpp_omp/sat_solver.cpp`
Script to run: `FinalProject/cpp_omp/run.sh`, `FinalProject/cpp_omp/slurm-build.sh`, `FinalProject/cpp_omp/slurm-run.sh`
Example commands to run in slurm:

- `$ cd FinalProject/cpp_omp/`
- Same as baseline

Please edit `slurmrun.sh`, if you want to change the default test case.
Example commands to run in your own machine:

- `$ cd FinalProject/cpp_omp/`
- `$ ./run.sh ../test_cases/flat30-13.cnf` 4
- Similar

5 Conclusions and Future Work

In conclusion, this study has delved into the realm of SAT problem-solving, with a specific focus on parallelization strategies to enhance the efficiency of solving complex instances. The implementation and comparative analysis of three parallelization approaches - `cpp_mpi`, `cpp_pthread`, and `cpp_omp` - have shed light on their respective strengths and optimal configurations. The results underscore the scalability of parallelization, particularly in handling larger problem sizes, with `cpp_mpi` demonstrating superior speed and efficiency in collaborative clause learning.

Looking ahead, there are several avenues for future work in this study. Firstly, the exploration of hybrid parallelization strategies that combine the strengths of multi-node and multi-threaded approaches could provide further performance improvements. Additionally, investigating adaptive parallelization mechanisms that dynamically adjust to the characteristics of specific SAT instances could lead to more efficient problem-solving.

References

[1] Roberto J Bayardo Jr and Robert Schrag. “Using CSP look-back techniques to solve real-world SAT instances”. In: *Aaai/iaai*. Citeseer. 1997, pp. 203–208.

[2] JP Marques Silva and Karem A Sakallah. “GRASP-a new search algorithm for satisfiability”. In: *Proceedings of International Conference on Computer Aided Design*. IEEE. 1996, pp. 220–227.