GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Parallelisation of different SAT-Solving-Algorithms using MPI

Frederik Hennecke, Pascal Brockmann

MatrNr: 21765841, 21227757

Supervisor: Zoya Masih

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2024

Abstract

A SAT (Boolean Satisfiability) solver is a computer program that determines whether a given Boolean formula is satisfiable or not. This report shows the implementation and performance analysis of four different SAT solvers, including Brute Force, DP (Davis-Putnam), DPLL (Davis-Putnam-Logemann-Loveland), and CDCL (Conflict-Driven Clause Learning). Each solver was also parallelized using MPI. We tested the solvers on standard benchmarks, comparing the sequential and parallel versions. Parallelization significantly improved performance for most of the implemented algorithms. The performance improvement is particularly noticeable with large input formulas.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work, I have used ChatGPT or a similar AI-system as follows:

- ☐ Not at all
- ☑ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure you that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Figures v

List of Abbreviations vi

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Objectives . . . 1
- 1.3 Structure . . . 1

2 Methodology 2
- 2.1 SAT Overview . . . 2
  - 2.1.1 Problem Definition . . . 2
  - 2.1.2 DIMACS Format Overview . . . 2
  - 2.1.3 NP and SAT . . . 3
  - 2.1.4 Applications of SAT . . . 4
- 2.2 Brute Force Solver . . . 4
- 2.3 Brute Force Parallelization . . . 5
- 2.4 DP (Davis-Putnam) Solver . . . 6
- 2.5 DP Parallelization . . . 7
- 2.6 DPLL (Davis-Putnam-Logemann-Loveland) Solver . . . 7
- 2.7 DPLL Parallelization . . . 9
- 2.8 CDCL (Conflict-Driven Clause Learning) Solver . . . 9
  - 2.8.1 CDCL Procedure . . . 9
  - 2.8.2 Example . . . 11
- 2.9 CDCL Parallelization . . . 12
  - 2.9.1 Portfolio-Based Approach . . . 12
  - 2.9.2 Initial Variable Decision and VSIDS Heuristic . . . 12

3 Implementation Details 12
- 3.1 Project Structure and Hosting . . . 12
- 3.2 Build System: CMake . . . 13
- 3.3 Testing: Googletest . . . 13
- 3.4 Input and Execution . . . 13
- 3.5 Sequential and Parallel Solvers . . . 13
- 3.6 Input Handling . . . 14

4 Experimental Setup 14
- 4.1 Test Cases . . . 14
- 4.2 Hardware Environment . . . 15

5 Performance Analysis 16
- 5.1 Brute Force Solver . . . 16
  - 5.1.1 SAT Cases . . . 16
  - 5.1.2 UNSAT Cases . . . 18
- 5.2 DP Solver . . . 19
- 5.3 DPLL Solver . . . 20

5.3.1 SAT Cases . . . 20
5.3.2 UNSAT Cases . . . 21
5.4 CDCL Solver . . . 22
5.4.1 SAT Cases . . . 22
5.4.2 UNSAT Cases . . . 23
5.5 Discussion . . . 24
5.5.1 Brute Force . . . 24
5.5.2 DP . . . 25
5.5.3 DPLL . . . 25
5.5.4 CDCL . . . 26
5.5.5 Comparison of the different algorithms . . . 26

6 Conclusion **26**
6.1 Summary of Findings . . . 26
6.2 Future Work . . . 27

References **28**

List of Figures

1 DPLL procedure . . . 8
2 CDCL procedure . . . 11
3 Runtime of brute force algorithm with 10 to 25 literals (all formulas satisfiable) . . . 16
4 runtime of brute force algorithm with 10 to 20 literals (all formulas satisfiable) 17
5 Runtime of brute force algorithm (all formulas not satisfiable) . . . 18
6 Runtime of DP algorithm with 1 to 36 variables . . . 19
7 Runtime of DPLL algorithm with 20 to 100 variables (all formulas satisfiable) 20
8 Runtime of DPLL algorithm with 50 to 100 variables (all formulas not satisfiable) . . . 21
9 Runtime of CDCL algorithm with 20 to 150 variables (all formulas satisfiable) 22
10 Runtime (in log scale) of CDCL algorithm with 20 to 150 variables (all formulas satisfiable) . . . 22
11 Runtime of CDCL algorithm with 20 to 125 variables (all formulas unsatisfiable) . . . 23
12 Runtime (in log scale) of CDCL algorithm with 20 to 125 variables (all formulas unsatisfiable) . . . 24

List of Abbreviations

MPI Message Passing Interface

CNF Conjunctive Normal Form

DIMACS Discrete Mathematics and Theoretical Computer Science

DP Davis-Putnam

DPLL Davis-Putnam-Logemann-Loveland

CDCL Conflict-Driven Clause Learning

SAT Satisfiable

UNSAT Not satisfiable

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

SCC Supercomputing Center

NHR National High-Performance Computing

VSIDS Variable State Independent Decaying Sum

1 Introduction

1.1 Background and Motivation

The Boolean Satisfiability Problem (SAT) is fundamental in computer science. Given a Boolean formula in conjunctive normal form (CNF), the SAT problem checks whether an assignment of values exists to the variables that satisfy the formula. Despite its importance, SAT is an NP-complete problem, meaning that the running time of traditional algorithms increases exponentially with the input size.

The first SAT solvers were developed in the 1960s, namely the Davis-Putnam procedure[DP60]. In the 1980s, more efficient algorithms, such as the DPLL (Davis-Putnam-Logemann-Loveland) algorithm, were developed.[ML62] In the 1990s, there were multiple papers[MS][MS99][BS97] about Conflict-Driven Clause Learning (CDCL) solvers, which are still widely used today.

In this report, we show the design, implementation, and evaluation of four parallel SAT solver algorithms: brute force, DP, DPLL, and CDCL. Each algorithm is first implemented sequentially and then parallelized using MPI. The parallel implementations will improve the runtime of several SAT-solving algorithms. The report describes the algorithms, implementation, and performance results, showing the benefits of parallel computation.

The project is open-source and can be seen in our Gitlab Repository.

1.2 Objectives

The primary objective of our project is to see how parallelization can improve the performance of SAT solvers. To achieve this, we implemented four different SAT solver algorithms sequentially. Then, we parallelized all four algorithms using MPI, so we have eight. We hope to achieve an increase in performance, which is approximately proportional to the number of processes. This means that if we compare a sequential (single process) algorithm to the parallel version of that algorithm with two processes using the same test formula in both cases, we want the parallel version, on average, to take roughly half the amount of time or less to find a solution than the sequential algorithm. This will not always be possible because the runtime of each algorithm can vary greatly, given different formulas. Furthermore, using MPI introduces some overhead to the algorithm, particularly with small test formulas. Additionally, we want to compare the performance of the four different algorithms to see if the more modern algorithms are faster.

1.3 Structure

This report is organized into the following sections:

- Section 2: Methodology provides an overview of the SAT problem and discusses the implementation of various solvers, including brute force, DP, DPLL, and CDCL. It also explains how these algorithms were parallelized using MPI

- Section 3: Implementation Details outlines the technical aspects of the project, including the tools and libraries used, and the structure of the codebase

- Section 4: Experimental Setup describes the hardware environment, and test cases used to evaluate the solvers.

- Section 5: Performance Analysis focuses on the results of the experiments, with a detailed analysis of the performance of the sequential and parallel solvers.
- Section 6: Conclusion summarizes the key findings and discusses potential areas for future improvements.

2 Methodology

2.1 SAT Overview

The Boolean Satisfiability Problem (SAT) is one of the most known problems in computer science and propositional logic. SAT involves determining whether an assignment of truth values (true or false) exists to a set of Boolean variables that satisfies a given Boolean formula. A formula is "satisfiable" if such an assignment exists and "unsatisfiable" otherwise.

2.1.1 Problem Definition

Formally, the SAT problem is defined as a Boolean formula expressed in Conjunctive Normal Form (CNF). A CNF formula consists of a conjunction (AND) of one or more clauses, where each clause is a disjunction (OR) of literals. A literal is either a Boolean variable or its negation.

For example, a CNF formula could look like:

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

In this case, the formula consists of three clauses:

1. $(x_1 \lor \neg x_2)$
2. $(x_3 \lor x_2)$
3. $(\neg x_1 \lor \neg x_3)$

The task is to find a truth assignment (e.g., $x_1 = \text{true}, x_2 = \text{false}, x_3 = \text{true}$) that makes the entire formula evaluate to true.

2.1.2 DIMACS Format Overview

In the DIMACS format, the CNF formula is represented as follows[Bur08]:

- Each variable is assigned a unique positive integer.
- A clause is represented as a sequence of these integers, where a positive integer indicates a variable and a negative integer indicates its negation.
- The formula is terminated by a zero (‘0‘).

A DIMACS file begins with metadata, followed by the clauses:

1. Header:

- The header starts with the letter *p* and is followed by *cnf*, the number of variables, and the number of clauses.
- For example, *p cnf 3 3* indicates a formula with three variables and three clauses.

2. Clauses: Each clause is written as a sequence of integers (representing the literals), ending with a *0*.

The CNF formula:

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

would be represented in DIMACS format as:

```
p cnf 3 3
1 -2 0
3 2 0
-1 -3 0
```

1. Line 1 ('p cnf 3 3'): Specifies that the formula has three variables and three clauses.
2. Line 2 ('1 -2 0'): Represents the clause $x_1 \lor \neg x_2$.
3. Line 3 ('3 2 0'): Represents the clause $x_3 \lor x_2$.
4. Line 4 ('-1 -3 0'): Represents the clause $\neg x_1 \lor \neg x_3$.

This simple and compact format allows SAT solvers to parse and operate on CNF formulas efficiently.

2.1.3 NP and SAT

In computational complexity, NP (Nondeterministic Polynomial time) refers to the class of decision problems for which a given solution can be verified in polynomial time. In simpler terms, if someone gives you a potential solution to a problem in NP, you can quickly (in polynomial time) check whether the solution is correct, even if finding that solution might take much longer.[KT05]

Formally: The complexity class of decision problems for which answers can be checked for correctness, given a certificate, by an algorithm whose run time is polynomial in the size of the input (that is, it is NP) and no other NP problem is more than a polynomial factor harder. Informally, a problem is NP-complete if answers can be verified quickly, and a quick algorithm to solve this problem can be used to solve all other NP problems quickly. [Bla21]

For example, given a CNF formula:

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

If we assume that $x_1 = true, x_2 = false, x_3 = true$, we can quickly verify that this assignment satisfies the formula by checking each clause:

$$x_1 \lor \neg x2 = true \lor true = true$$
$$x_3 \lor x2 = true \lor false = true$$
$$\neg x_1 \lor \neg x3 = false \lor false = false$$

The formula is not satisfied in this case, but this verification process only took linear time regarding the number of clauses and variables. NP-Completeness and Cook's Theorem

In 1971, Stephen Cook introduced Cook's Theorem, which proved that the SAT problem is NP-complete, meaning that:

SAT is in NP (as we can verify solutions in polynomial time). SAT is at least as hard as any other problem in NP.[Coo71]

Since SAT was the first problem proven NP-complete, it is central to complexity theory. Many real-world problems, from optimization to scheduling to verification, can be reduced to SAT, making SAT solvers critical in addressing NP-complete problems.

2.1.4 Applications of SAT

SAT solvers are widely used in various fields, including, but not limited to:

- Formal Equivalence Checking: SAT solvers check whether two circuit representations (e.g., a specification and its implementation) are functionally equivalent. This may ensure that the design transformation or optimization process has not introduced any errors.[BGV99]
- Model Checking: SAT solvers help verify whether a given model of a system (e.g., a circuit or software system) satisfies certain properties (typically safety or liveness properties). This is important in the design and verification of complex hardware systems.[BGV99]
- Formal Verification of Pipelined Microprocessors: Microprocessors often employ pipelining for efficiency, but this introduces complex interactions between different stages. SAT-based formal verification methods are used to ensure the correctness of these designs under various conditions, including instruction reordering and hazards.[BGV99]
- Automatic Test Pattern Generation (ATPG): SAT solvers can help generate test patterns that detect faults in digital circuits, enabling efficient fault detection during manufacturing testing.[NSR02]
- Routing of FPGAs: SAT solvers assist in routing Field-Programmable Gate Arrays (FPGAs) by solving constraint satisfaction problems related to connecting different components while satisfying the design's constraints on timing, signal integrity, and layout.[NSR02]
- Planning and Scheduling Problems: SAT solvers also solve combinatorial problems such as planning and scheduling. One such application would be the solver in conda.[24b]

Given its wide-ranging applications, solving SAT problems is crucial in theoretical and practical domains.

2.2 Brute Force Solver

The basic idea of the brute force algorithm is to check for each possible assignment of variables to see if it satisfies the formula. This means that if we have n variables, we have

to check up to 2^n possible assignments since each variable can have two different values: "true" and "false." The maximum number of necessary checks occurs when the formula is not satisfiable or when only the last checked assignment of the variables satisfies the formula (in our case, all variables are set to "true").

For our implementation, we create a boolean vector of length n (number of variables) and set every vector entry to "false." This vector represents the current assignment of variables, with each vector entry representing a single variable. By doing this, we can easily represent the different assignments. To go through every possible assignment of the variables, we treat the assignment vector as an n-bit binary number. This binary number is increased by one after each satisfiability check if the current assignment of variables does not satisfy the formula. If an assignment satisfies the formula, "true" will be returned, and the brute force procedure will stop since a solution was found. To see if all possible assignments of variables were tested, we check if all entries of the assignment vector are set to "true." If that is the case, every possible number from 0 to $2^n - 1$ and, therefore, every possible assignment of variables was tested, and "false" can be returned since the formula is not satisfiable.

Since the clauses of the formula are connected via conjunctions, every clause must be "true" for the formula to be "true." To test if a clause is satisfied, we check whether the clause contains a variable set to "true" or the negation of a variable set to "false." Since the clauses only contain disjunctions and no conjunctions, it is enough for a single literal to be "true." To check whether the entire formula is "true," we count how many clauses satisfy the current assignment of variables. A solution is found if the number of satisfied clauses equals the number of clauses of the formula.

2.3 Brute Force Parallelization

In order to have multiple processes work on the satisfiability check, we divide the variable assignments, which have to be tested, into m blocks (partitions), with m being the number of parallel processes. We return to the binary number vector from the sequential version to do this. First, we calculate how many different numbers we can depict using an n (number of variables) bit binary number, which is 2^n. Then, we divide this number by m, the number of parallel processes, so we know how many assignments of variables each process has to go through (partition size). To make each process work on a different partition, we give each process a start and stop value for their respective partition. The start value is the rank of the process(from 0 to the total number of processes) of the process times the partition size, while the stop value is the start value plus the partition size. The next step is to calculate those numbers' binary representation and write that representation into the binary number vector. The check, whether the formula is "true" or "false" with the current assignment of variables, works like the sequential algorithm. If a solution satisfies the formula, the process that found the solution will send a stop signal to the other processes. Each process uses a nonblocking MPI Receive to check for this stop signal. The synchronization routine MPI Barrier is used to wait for each process to synchronize to that specific point. Finally, the solutions of the different processes are synchronized using a blocking MPI Reduce. Since each process works on a different partition, each process has to reach the result "false" in order for the formula to be not satisfiable.

Our implementation has a limitation, however. Since we are using 64 bits to save the number of possible variables assigned, we can only work with up to 64 variables.

Therefore, we have to check if the number of variables is greater than 64 and throw an error if that is the case. This, however, is fine since both brute-force algorithms are pretty slow. Because of that, we limit the maximum number of variables in the performance analyses for both brute force algorithms.

2.4 DP (Davis-Putnam) Solver

The Davis Putnam (DP) algorithm is based on resolution.[DP60] The resolution rule states: If we have a formula in conjunctive normal form (CNF), we can create a new clause, which is implied by two "parent" clauses. If we have two clauses C_1 and C_2 with C_1 containing a variable A and C_2 its negation $\neg A$, the new clause C_{new} can be determined by $C_1/A \vee C_2/\neg A$. In Other words: The new clause C_{new} is a disjunction of the clause C_1 without the variable A and the clause C_2 without the negation of variable A.[Bec06]

Our implementation of the DP algorithm is recursive. Every recursion step will go through the following parts in order: three satisfiability checks for clauses and literals, a check for the satisfiability of the entire formula, and the actual DP procedure.

In order to check for the satisfiability of the entire formula, clauses that are determined to be "true" will be removed from the formula. By doing this, we can easily test if the entire formula is "true" by checking if the formula vector is empty since every clause was determined to be "true." We also remove literals from a clause if that literal is determined to be "false." This means if we find an empty clause vector, that clause is "false" since all of its literals are "false." Therefore, the entire formula is "false" because all clauses are connected via conjunctions. These are the methods used to check for the satisfiability of the entire formula (stopping conditions for recursion).

There are three satisfiability tests for clauses and literals. The first one is the unit propagation. This checks for unit clauses. A unit clause is a clause that only contains one literal. The algorithm does this by simply checking the length of each clause vector. If a unit clause is found, its single literal must be "true," so the clause is "true" as well. This means each clause containing that literal can be removed since the literal has to be "true"; therefore, each clause containing that literal is "true." Furthermore, each occurrence of the negation of that literal is removed as well since the negation of that literal will always be "false." The next step is to eliminate clauses that are not in proper CNF. This means we delete every clause containing a variable and its negation since this clause will always be "true." The last part of the satisfiability tests for clauses and literals is the elimination of pure literals. This means we check whether or not each variable occurs in both polarities. Suppose we find a variable that only occurs in one polarity. In that case, that variable can be set to "true" if its polarity is positive or to "false" if its polarity is negative without resulting in a literal or clause becoming "false." This means we can simply remove each clause containing a variable that only occurs in one polarity.

The final step is the actual DP procedure. For simplicity, our implementation chooses the first literal of the first clause for the resolution. Then, it uses the resolution rule to create new clauses by combining one clause containing the chosen literal with a clause containing the negation of the chosen literal and adding them to the formula. This happens for every possible pair of a clause containing the chosen literal and a clause containing the negation of the chosen literal. This means that if we have n clauses containing the chosen literal and m clauses containing the negation of that literal, we will add $n \cdot m$ new clauses. The last action before starting the next recursion step is to remove all clauses containing the chosen literal or its negation. This means n plus m clauses will

be removed.

2.5 DP Parallelization

In order to parallelize the DP algorithm, each process uses the previously explained DP algorithm, but each chooses different variables for the DP procedure (resolution). To choose a variable for the resolution, first, each process compares its process rank with the current size of the formula vector minus one. The smaller value of the two will be chosen, and we will call it n as an example. Then, the first variable of the n-th formula will be chosen as the variable for the resolution. The comparison between process rank and the actual size of the formula array is necessary. Otherwise, the algorithm might try to work with an index outside the formula vector's current index range since the process rank can be greater than the actual size of the formula array.

If a process reaches a result, whether "true" or "false," it will send a stop signal to each other process. Each process uses a nonblocking receive to check for this stop signal and will abort if the stop signal is received. The synchronization routine MPI Barrier is used to wait for each process to stop its algorithm. Finally, the solutions of the different processes must be synchronized using a blocking MPI Reduce.

2.6 DPLL (Davis-Putnam-Logemann-Loveland) Solver

The Davis Putnam Logemann Loveland (DPLL) algorithm is a satisfiability check algorithm based on backtracking. Its basic idea is similar to the brute force algorithm. However, instead of assigning all variables at once, the variables will be assigned a value one after another with multiple satisfiability checks in between.[ML62] This way, not all 2^n (n is the number of variables) possible variable assignments must be tested. If, for example, setting the first two variables to "true" is enough to determine that the formula is "false," every variable assignment with the first two variables set to "true" will result in the formula being "false." Therefore, the algorithm does not need to continue this path and can reassign one of the first two variables to "false" instead.

The idea is to set one variable to "true" and then run through multiple satisfiability checks. If these checks' results are "true," a solution is found, and the algorithm can stop. If it is not yet possible to tell if the formula is "true" or "false" with the current assignment of variables, another variable will be set to "true." This way, the number of assigned variables will increase with each loop until we have to backtrack. If the result is "false," the algorithm will backtrack to the state in which the last variable, which is currently "true," was assigned, set it to "false" instead, and then continue with the satisfiability checks. If the result of the satisfiability checks is "false" and every assigned variable is already set to "false," the algorithm will stop and return "false" since there is no assignment of variables, which satisfies the formula. A pictorial representation of this procedure can be seen in Figure 1.

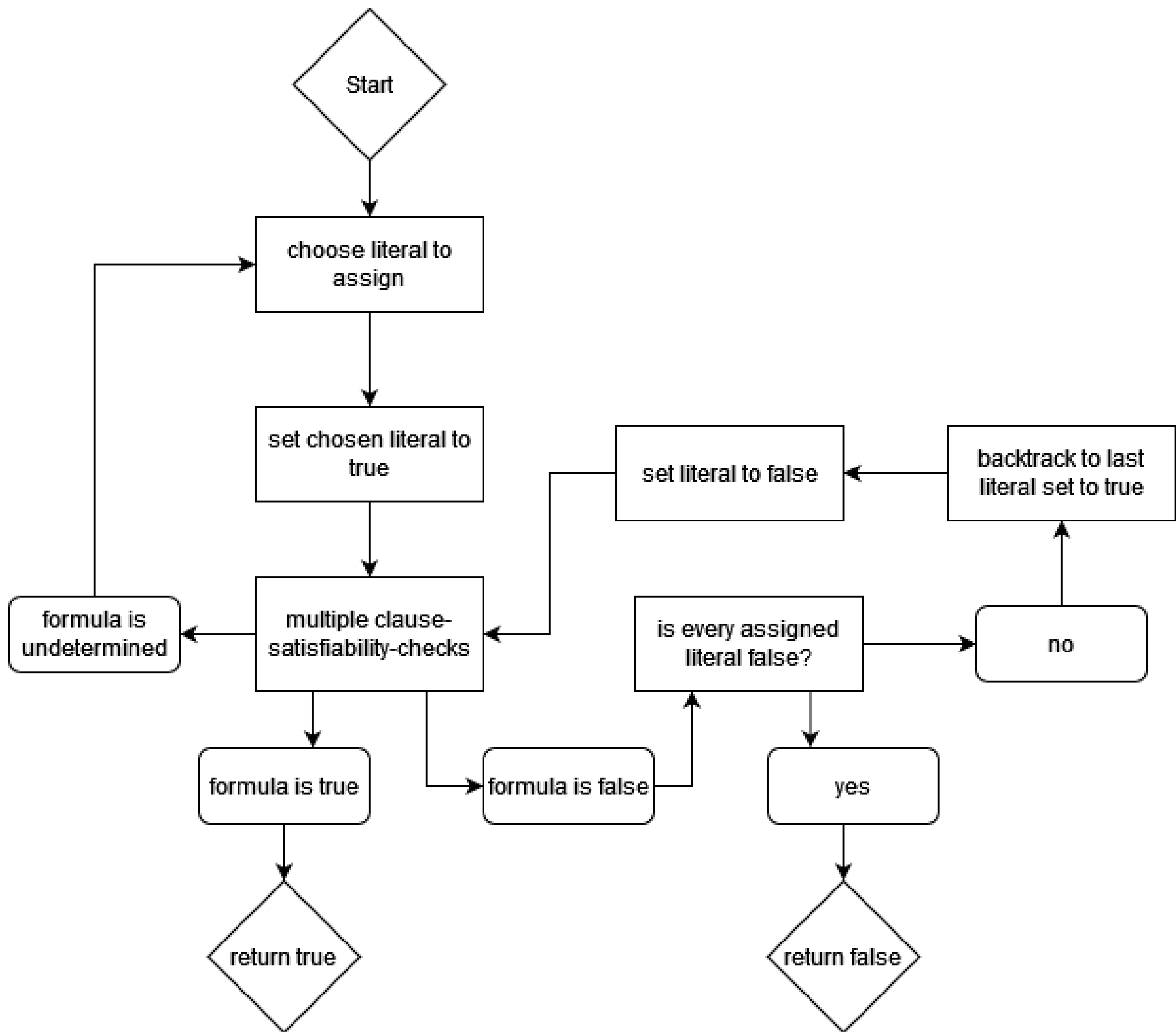

Figure 1: DPLL procedure

Our implementation of the DPLL algorithm is recursive and similar in structure to our implementation of the DP algorithm, but it uses an additional vector of length n (number of variables). This vector is used to track the current assignment of the variables. Each variable can have three states: "true" (1), "false" (-1) and undetermined (0). In the beginning, each variable is set to undetermined. This vector and the formula vector are the parameters for the DPLL procedure and are given to it via call-by value. This way, each recursion step works on its own vectors, and we have copies for each step if backtracking is required. For this reason, the DPLL algorithm takes up a comparatively large amount of RAM. In order to save some memory, the formula and the assignment vector are of the type short (16 bits) and not int (32 bits). The entire procedure consists of multiple satisfiability checks for clauses and literals, a check for the satisfiability of the entire formula, and finally, the actual DPLL procedure.

The formula check (stopping condition) works like in the DP algorithm. This means we will again remove each clause that is "true" from the formula vector and each literal that is "false" from the clause vectors. So we can check if the formula vector is empty, which means the formula is "true," or if a single clause vector is empty, which means the formula is "false."

Again, there are three satisfiability checks for clauses and literals. This time, though, the first is the assignment check. This function checks for the current variable assignment, which clause is "true" and which literal is "false," and removes them from the formula vectors or clause vectors, respectively. The other two clause- and literal checks are unit propagation and pure literal elimination. Both were already explained for the DP algorithm and worked the same in the DPLL algorithm. This time, however, both also change the state of the corresponding variable in the assignment vector.

For the DPLL procedure, we first have to find the next variable that is still undetermined, set it to "true" (1), and start the next recursion step. The order in which the variables are assigned is from smallest to highest index of the variable vector. Because the parameters are given to the recursive function call via a call by value, the current state of both vectors will be kept for the current recursion step. It can be used, if backtracking is necessary, by changing the variable we set to "true" (1) in this recursion step to "false" (-1) instead and stating the next recursion step again. If this also returns 0, the entire DPLL algorithm will return 0.

2.7 DPLL Parallelization

In order to parallelize the DPLL algorithm, we use the same basic idea that we used to parallelize the brute force algorithm: Dividing the variable assignments into multiple partitions. This time, we accomplish this by setting the first few variables to "true" or "false" in the assignment vector before the actual DPLL algorithm begins. The number of assigned variables depends on the number of processes. Each process will be given a different assignment vector. For example, if we have two processes, the first will start with the first variable set to "true," while the second process will start with the same variable set to "false." After that, each process will work the same way the sequential DPLL algorithm did. It is important to note that the variables, which were set before the actual DPLL algorithm started and which differentiate the partitions from one another, are not changed by the algorithm.

A nonblocking receive is used to check for a stop signal, which will be sent if a process finds an assignment of variables that satisfies the formula. The synchronization routine MPI Barrier waits for each process to finish, and a blocking MPI Reduce synchronizes the results. If the formula is not satisfiable, each process has to reach the result "false" since all of them are working on different partitions.

For our implementation of the parallel DPLL algorithm, the number of processes must be a power of two because the number of variables set before the actual DPLL algorithm must be equal for each process.

2.8 CDCL (Conflict-Driven Clause Learning) Solver

2.8.1 CDCL Procedure

The CDCL algorithm is based on the core concepts of DPLL but extends it with conflict-driven techniques. The solver iteratively assigns truth values to variables, performs unit propagation to simplify the formula, detects conflicts, analyzes the cause of conflicts, learns new clauses to avoid repeating mistakes, and backtracks. What sets CDCL apart is its ability to learn from conflicts and optimize future decisions based on previous conflicts, making it far more efficient in pruning the search space.

CDCL was the work of many different papers (e.g., [MS][MS99][BS97]) and is still being improved today.

The CDCL algorithm operates as follows:

1. Initialization and Decision Making: The solver starts by initializing the problem (e.g., reading input) and setting up data structures like the watchlist, which tracks two literals from each clause for efficient propagation. CDCL uses decision heuristics such as Variable State Independent Decaying Sum (VSIDS) to guide the variable selection process. VSIDS assigns scores to variables based on their occurrences in conflicts, with each conflict increasing the score. This helps focus the search on variables more likely to cause a conflict.

2. Unit Propagation: After making a decision (assigning a truth value to a variable), the solver propagates this assignment through the formula using unit propagation. This step ensures that any clause with only one unassigned literal force is set to true, simplifying the formula. CDCL uses the two-literal watching scheme to track clause satisfaction without revisiting every clause. A conflict is detected if any clause becomes unsatisfiable (all literals are false).

3. Conflict Detection and Analysis: When a conflict is encountered (a clause becomes unsatisfiable), the solver starts the conflict analysis. This involves examining the cause of the conflict and generating a learned clause—a new clause that records the circumstances leading to the conflict. The learned clause helps prevent the solver from making the same conflicting assignments in the future, thereby pruning the search space.

4. Clause Learning and Non-Chronological Backtracking: One of the strengths of CDCL is its ability to perform non-chronological backtracking, also known as back-jumping. Unlike DPLL, which backtracks in a fixed order, CDCL can jump back to a decision level directly responsible for the conflict. This reduces the number of assignments the solver needs to reconsider. The learned clause is added to the formula, which can help the solver ensure it avoids repeating the same conflict in future searches.

5. Heuristics and Restarts: The VSIDS heuristic also undergoes periodic decay, which reduces the scores of older variables and helps the solver focus on more recent conflicts. In addition to this decision-making, CDCL solvers use a restart strategy to prevent the solver from getting stuck in unproductive parts of the search space. In this case, we decided to use the Luby sequence[LSZ97], which controls when restarts occur by following a predefined sequence of intervals. Restarts allow the solver to reconsider earlier decisions with the benefit of newly learned clauses. These restarts happen after a specified amount of iterations, namely

$$t_i = \left\{ \begin{array}{ll} 2^{k-1} & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1}, & \text{if } i = 2^k - 1 \leq I < 2^k - 1 \end{array} \right\}$$

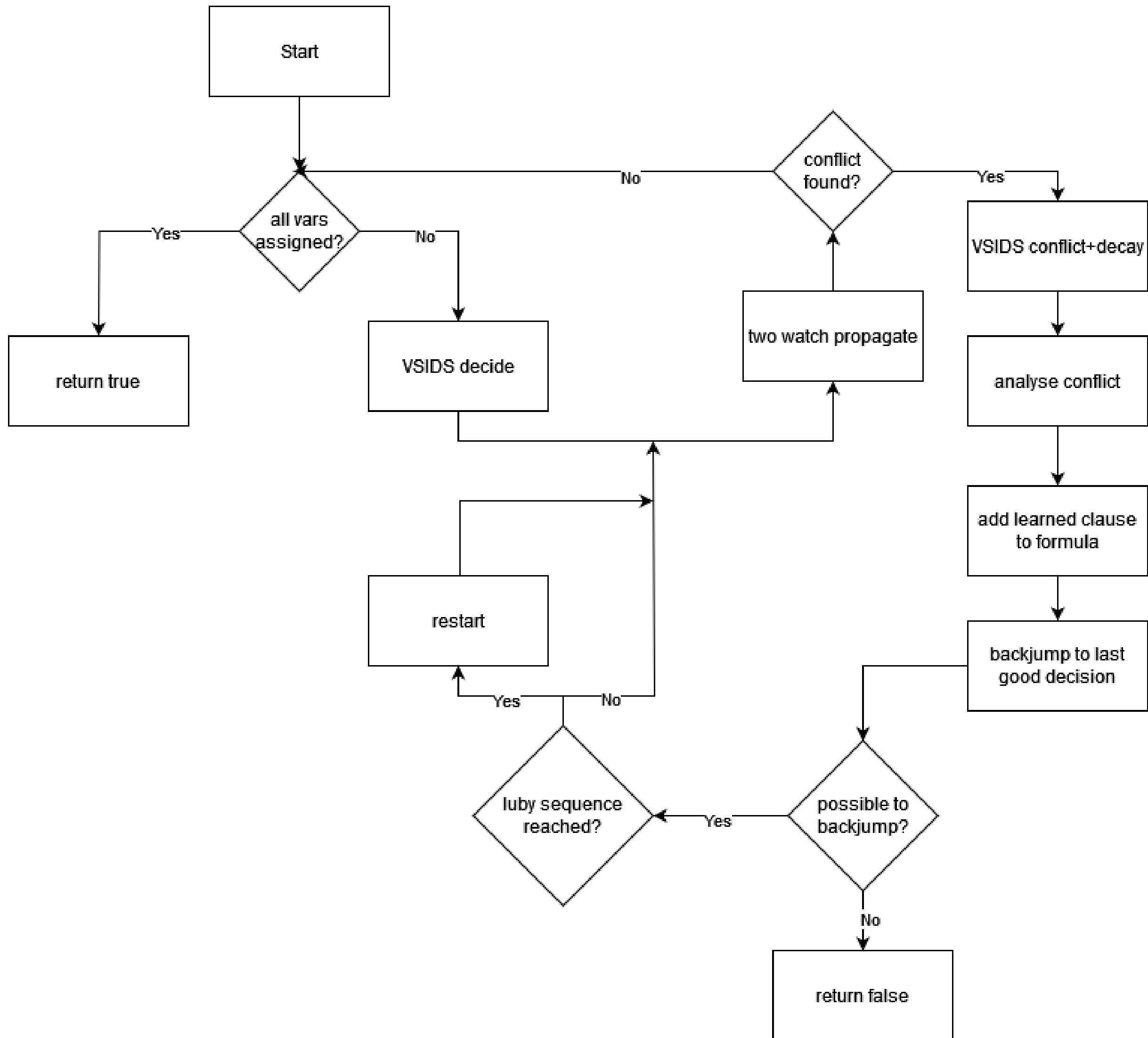

Figure 2: CDCL procedure

2.8.2 Example

Consider a SAT problem with the formula $(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (\neg x_3 \lor x_2)$. The CDCL solver begins by deciding, assigning $x_1 =$ true. This assignment triggers *unit propagation*: since x_1 satisfies the clause $(x_1 \lor \neg x_2)$, no further action is needed there, but the clause $(\neg x_1 \lor x_3)$ forces $x_3 =$ true. Now, the clause $(\neg x_3 \lor x_2)$ requires $x_2 =$ true, completing unit propagation.

If a conflict occurs later (e.g., a future assignment contradicts the current set), CDCL performs *conflict analysis*, identifying the combination of assignments that led to the conflict. It then learns a new clause, preventing the same conflict from happening again, and backtracks to a relevant decision point. After learning, CDCL may restart the process with the learned clause now part of the formula, refining the search path and improving efficiency. This decision-making cycle, propagation, conflict learning, and backtracking continue until the formula is satisfied or proven unsatisfiable.

2.9 CDCL Parallelization

2.9.1 Portfolio-Based Approach

In our parallel implementation, each MPI instance has its own CDCL solver. These solvers all operate independently, making separate decisions and applying clause learning locally within each process. Since the processes do not share information during execution, they avoid the synchronization overhead. The solvers will send messages to each other to terminate each process as soon as one finds a satisfying assignment or determines that the problem is unsatisfiable. Like the other approaches we chose, each solver instance checks with a non-blocking MPI Receive whether the other solvers are already finished. Finally, we use MPI Reduce is used to synchronize the results.

2.9.2 Initial Variable Decision and VSIDS Heuristic

Each process chooses its first decision variable differently to introduce variability among the solvers and improve search space coverage. We use the VSIDS heuristic to rank variables by their relevance based on conflict activity. However, rather than using the most important variable across all processes, each process selects a different starting variable based on rank.

- The VSIDS heuristic ranks the literals according to their importance. In the first iteration of the solver, it is just the total occurrences of each literal.
- Each process (ranked 0 through n) selects the n-th most important variable according to VSIDS for its first decision. For example, the first process chooses the most frequently occurring, the second process the second most frequently occurring, and so on.

By distributing the initial variable selection, we ensure that each solver begins exploring a different search space region, mostly avoiding overlap, except for a few edge cases. As each process proceeds, it uses the regular CDCL strategy—unit propagation, conflict analysis, clause learning, and backtracking—guided by the VSIDS heuristic. This variation in initial decisions improves the likelihood of one solver finding the solution faster than if all solvers used the same decision sequence. Especially with smaller inputs, some solvers can process the input the same as the others, but the chance of this happening decreases with the input size.

3 Implementation Details

3.1 Project Structure and Hosting

The entire project is hosted on GitLab. The source code is written in C++ due to its performance capabilities and the availability of valuable libraries for parallel processing.

The project is organized into several directories:

- *src/*: Contains the main solver implementations and supporting code.
- *src/Paral/*: Contains the parallel solver implementations using MPI.

- *src/Sequ/*: Houses the sequential solver implementations.
- *test/*: Includes test input files and unit test cases.

3.2 Build System: CMake

CMake is the build system that handles all dependencies and manages the compilation process. It automates building the project across different environments and platforms. The script worked in multiple environments. We tested it on Windows 10/11, Ubuntu 20.04, and Scientific Linux 7.9. The MPI implementations we used were OpenMPI, Microsoft MPI, and Intel MPI.

To build the project, users can run the following commands:

```
mkdir build
cd build
cmake ..
make
```

3.3 Testing: Googletest

The *test/* folder contains all the test input files and a unit test suite written using Googletest. This tests all of the solvers to ensure their correctness. The unit tests check for expected outputs for small SAT instances and help maintain the integrity of the codebase during development.

For example, running the unit tests involves executing the test suite, which can be done with the following command after building:

```
  ./testWithLibs
```

3.4 Input and Execution

The solvers are designed to be invoked from the command line. The *main file* in the *src/* folder handles the input and delegates it to the appropriate solver (sequential or parallel) based on the command-line arguments provided. The program reads the input directly from a *.cnf* file or the terminal.

The following is an example of how the program can be executed:

```
./SATMPI --algorithm cdcl --file ./test/input.cnf
```

Here, the user specifies the algorithm to use (in this case, CDCL) and provides the path to the input file. Alternatively, if input is provided via the terminal, the program will process the data directly from *stdin*. While the solvers themselves are parallelized, input reading is not parallelized due to the small size of typical input files, and the overhead of parallelizing file reading was deemed unnecessary.

3.5 Sequential and Parallel Solvers

The project is divided into two main solver categories:

- *Sequential Solvers*: Located in the *Sequ/* folder, these solvers implement the brute force, DP, DPLL, and CDCL algorithms without any parallelization.

- *Parallel Solvers*: Housed in the *Paral/* folder, these solvers use MPI to enable parallel processing.

As outlined in the methodology section, each process in the parallel solver selects its initial decision variable differently. MPI handles inter-process communication and process management, though direct communication between processes is minimized to avoid overhead, and solvers operate largely independently until one reaches a conclusion.

3.6 Input Handling

As mentioned earlier, input reading is done sequentially, even in the parallel versions. Since SAT problem files are relatively small, parallelizing input reading would add unnecessary complexity and overhead. Thus, each process reads the input independently.

4 Experimental Setup

4.1 Test Cases

The test cases used in this experiment were selected from two main sources to ensure a diverse set of SAT problems that could effectively demonstrate the capabilities and limitations of the solvers.

1. Uniform Random-3-SAT Problems:

    - We obtained test cases from the *SATLIB benchmark suite*[1], specifically from the *Uniform Random-3-SAT* category. These cases are standard in SAT research and are known to be challenging for SAT solvers.
    - Uniform Random-3-SAT is a family of SAT problem distributions generated by randomly constructing 3-CNF formulae with n variables and k clauses. Each clause is formed by randomly selecting three literals from the 2^n possible literals, with each literal selected with equal probability. The distribution can be divided into "unforced" and "forced" uniform Random-3-SAT. Unforced instances are generated without constraints, while forced instances are generated with a predetermined variable assignment that satisfies the formula. The distribution exhibits a phase transition phenomenon, where the probability of generating a satisfiable instance drops sharply at a critical number of clauses ($k' \approx 4.26n$) for large n. This phase transition region is particularly interesting for evaluating SAT algorithms, as instances from this region tend to be hard to solve for current SAT solvers.[CKT91]
    - Each test case in this dataset is labeled as either SAT (satisfiable) or UNSAT (unsatisfiable), allowing for verification of solver accuracy.
    - Test case sizes (i.e., the number of literals) were selected based on the complexity of the algorithm being tested. For example, the brute force solver was tested with smaller SAT instances due to its exponential complexity. In contrast, CDCL could handle larger instances due to its more advanced solving strategy.

[1] SATLIB benchmark suite

2. Pigeonhole Principle Problems:

    - We also generated test cases using *cnfgen* with the Pigeonhole Principle formula family[2]. This class of problems is notoriously difficult for certain types of SAT solvers, making it a good candidate for performance testing.[Hak85]

These test cases were used across all solvers (brute force, DP, DPLL, and CDCL), adjusting instance sizes depending on the respective solver's capabilities.

4.2 Hardware Environment

All experiments were executed on the GWDG Supercomputing Center (SCC), using resources provided by the NHR (National High-Performance Computing) infrastructure. The tests were run on the *medium partition*, which provided the following specifications[24a]:

- Nodes: 94
- CPU: 2 × Intel Cascade Lake 9242
- RAM: 364 GB per node
- Cores: 96 per node
- SSD: Yes

The parallel solvers were executed with varying numbers of nodes to analyze the performance as the solver operates on multiple cores.

[2] CNFgen Pigeonhole Principle

5 Performance Analysis

5.1 Brute Force Solver

5.1.1 SAT Cases

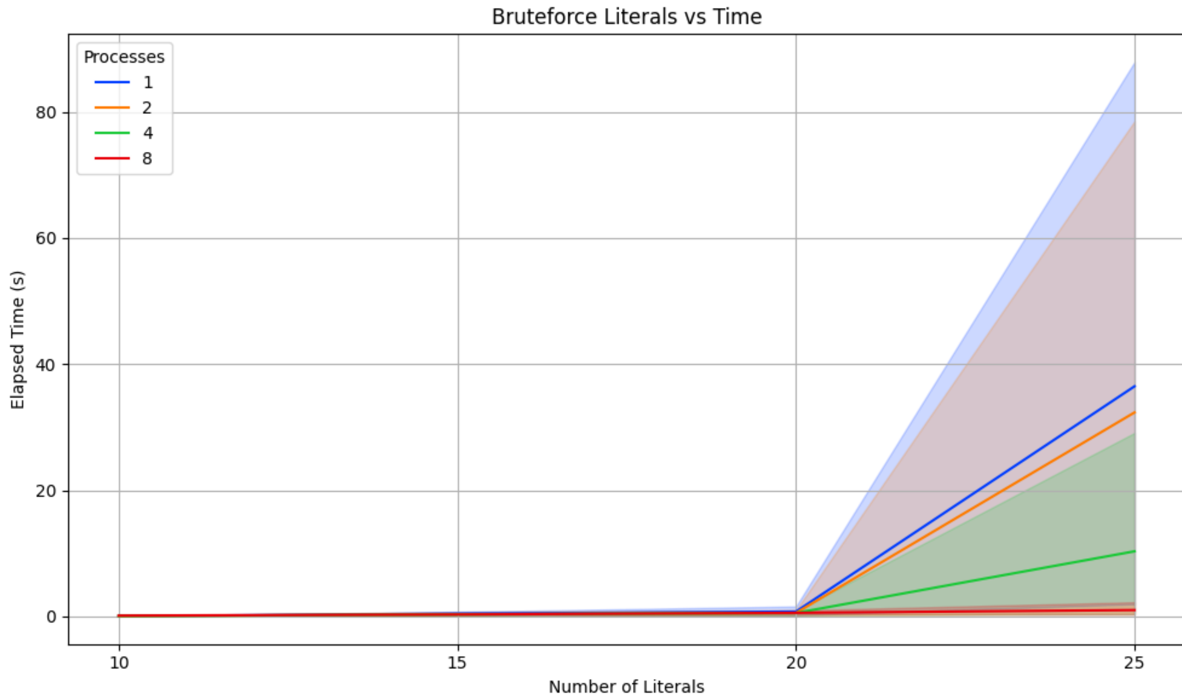

Figure 3: Runtime of brute force algorithm with 10 to 25 literals (all formulas satisfiable)

Figure 3 shows the runtime of the brute force algorithm with different numbers of processes (blue = sequential, orange = parallel with two processes, green = parallel with four processes, Red = parallel with eight processes) in seconds for satisfiable formulas. Each formula has twice as many clauses as literals (for example, the formulas with 25 literals have 50 clauses). Ten different formulas were used for each number of literals. The line shows the average runtime, while the shadow shows the deviations.

The first thing to note is that the runtime average and deviation increase drastically when the number of literals exceeds 20. The high deviation is not surprising, though, since the brute force algorithm can get "lucky" and hit an assignment of variables that satisfies the formula early or even on the first try. However, hitting that assignment at the 2^nth try is also possible. Furthermore, we can see that adding more processes decreases the average runtime and also the amount of deviation. This, again, is not surprising. More processes mean that the algorithm can test the possible variable assignments faster, and the likelihood of hitting a variable assignment that satisfies the formula early increases as well.

Interestingly, the runtime for formulas with less than 20 literals barely differs. Figure 4 shows the same data as Figure 3, but we focus on the formulas with 10 to 20 literals.

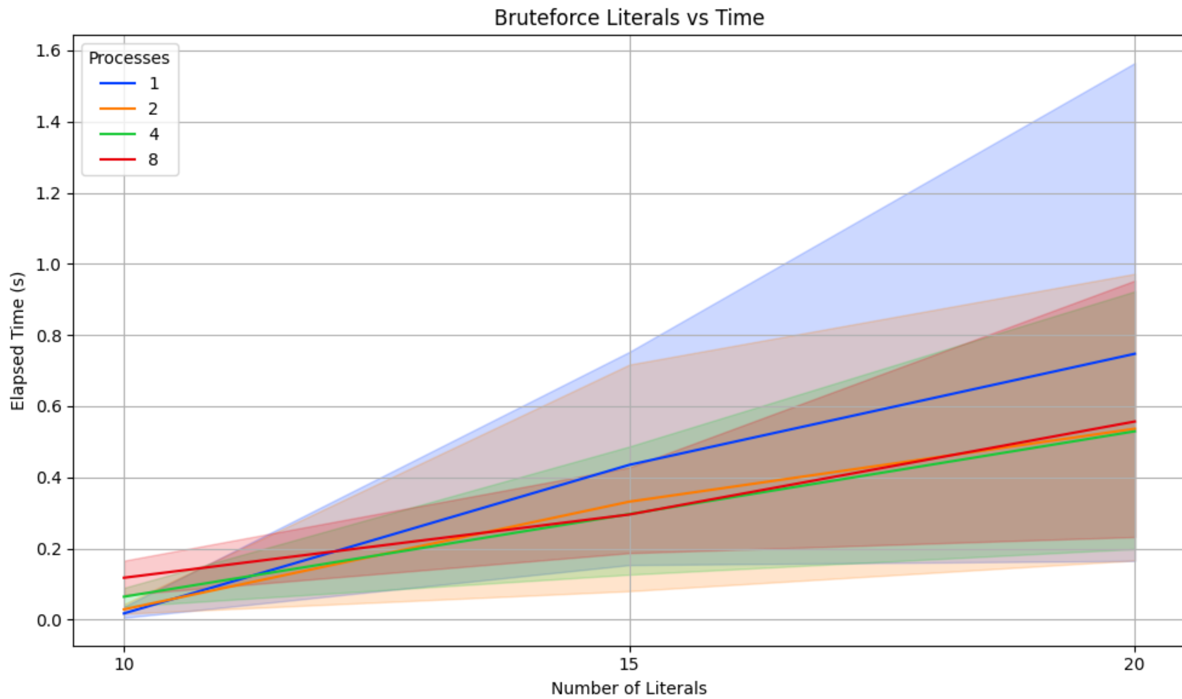

Figure 4: runtime of brute force algorithm with 10 to 20 literals (all formulas satisfiable)

The runtimes here barely differ, and the deviation could be higher. All averages are below one second, and even the time between the lowest and highest outliers is less than two seconds. On average, the parallel algorithms are faster than the sequential ones again. However, the parallel algorithms' runtime seems to be equal. This can at least partly be explained by the process of "getting lucky." Moreover, suppose one variable assignment that satisfies the formula is greater than $2^n/2$ and simultaneously close to it. In that case, each parallel algorithm will reach this solution in roughly the same amount of time since each multi-process algorithm has one process, which starts at $2^n/2$ because the number of processes is always a power of 2. So, to better understand the potential performance increase of the brute force algorithm resulting from parallelization, we will now look at the worst case for this algorithm: the formulas being not satisfiable.

5.1.2 UNSAT Cases

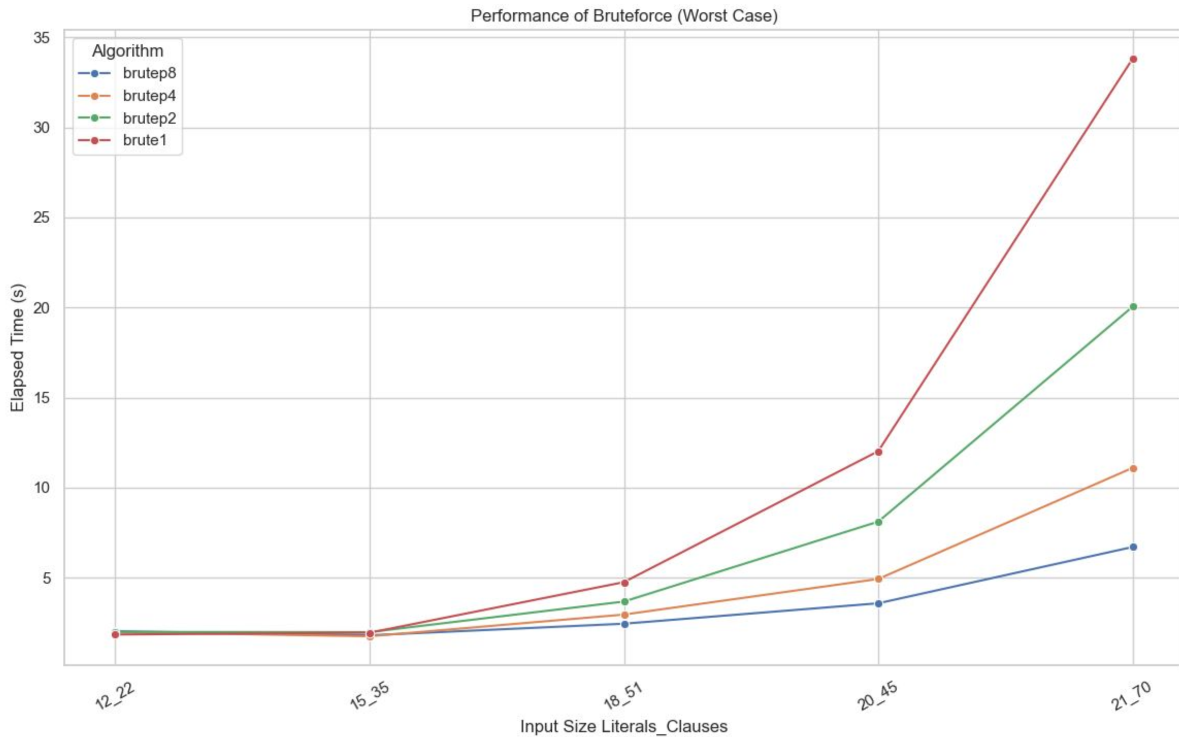

Figure 5: Runtime of brute force algorithm (all formulas not satisfiable)

Figure 5 shows the runtime of the brute force algorithm with different numbers of processes (red = sequential, green = parallel with two processes, orange = parallel with four processes, blue = parallel with eight processes) in seconds for not satisfiable formulas. This is the worst case for the brute force algorithm because it has to go through every possible variable assignment to reach the result. The chart also shows the size of the test formulas (for example, 12_22 means a formula of 12 variables and 22 clauses).

The runtime of the algorithm decreases when more processes are added. However, the performance difference is negligible when the test formulas are small. When the formula size increases, so does the general runtime and runtime, as well as the speed difference between the different brute-force algorithms. We can also see that the runtime for not satisfiable formulas is greater than that for satisfiable formulas of similar size. This is because the number of necessary variable assignments for each not satisfiable formula is 22^n, with n being the number of variables in the formula. At the same time, they usually do not need to check each possible variable assignment to see if the formula is satisfiable.

When we increase the number of processes by a factor, we roughly divide the runtime by the same amount (double processes => half runtime). Therefore, the runtime is roughly inversely proportional to the number of processes. This, however, is not surprising since the test formulas used are all not satisfiable (worst case). This means that when the formula has nn variables, each algorithm has to go through 2^n variable assignments. However, for the multi process algorithms, each process only has to go through a fraction of these. When m is the number of processes, each only has to go through $2^n/m$ variable assignments. S, and since the m processes work simultaneously, the runtime is also roughly divided by the number of processes m. The runtime is, however, not precisely divided

by the number of processes. This is caused by the process of parsing and building the formula, and assignment vectors are considered to be taken into account when checking the runtime. MPI also seems to add an overhead, which explains the small to nonexistent difference in runtime for very small test formulas.

5.2 DP Solver

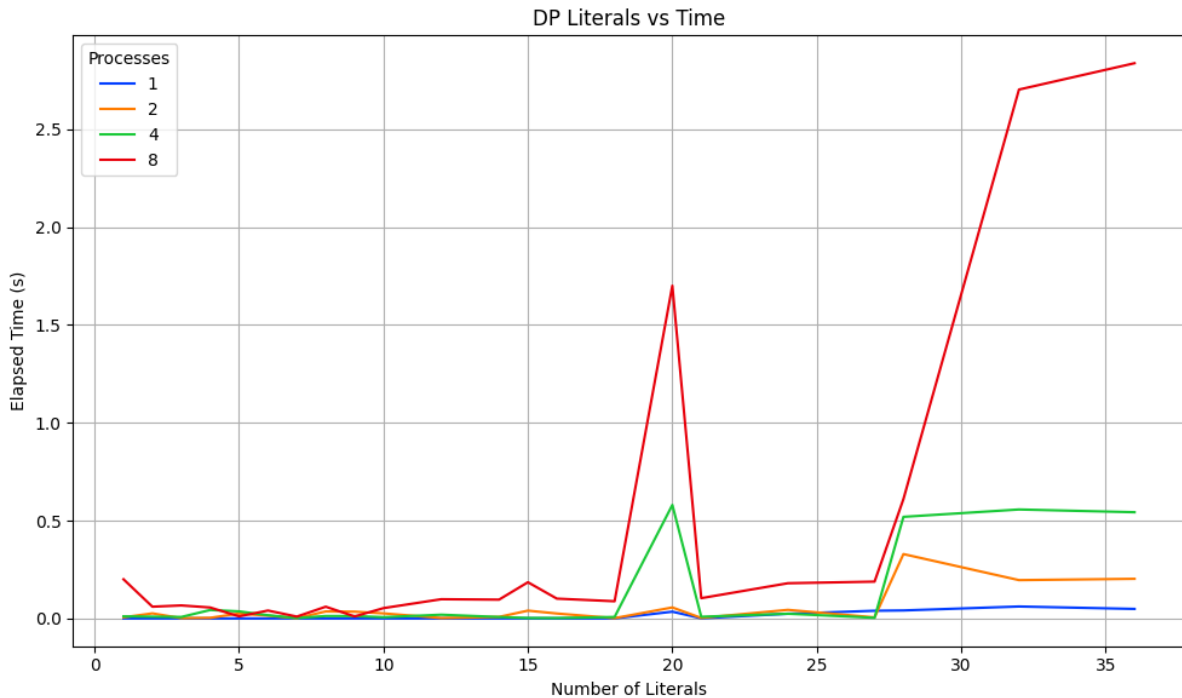

Figure 6: Runtime of DP algorithm with 1 to 36 variables

Figure 6 shows the average runtime of the DP algorithm with different numbers of processes (blue = sequential, orange = parallel with two processes, green = parallel with four processes, red = parallel with eight processes) in seconds. The number of clauses varied for each number of variables. Some of these formulas were satisfiable, while others were not satisfiable.

The average runtimes shown in Figure 6 are still relatively short (under 3 seconds), and most of the time, all four algorithms reached similar runtimes. However, the multi-process algorithms have some runtime spikes at random formula sizes. Furthermore, these spikes become more pronounced when the number of processes increases. Moreover, adding more processes increases the average runtime.

5.3 DPLL Solver

5.3.1 SAT Cases

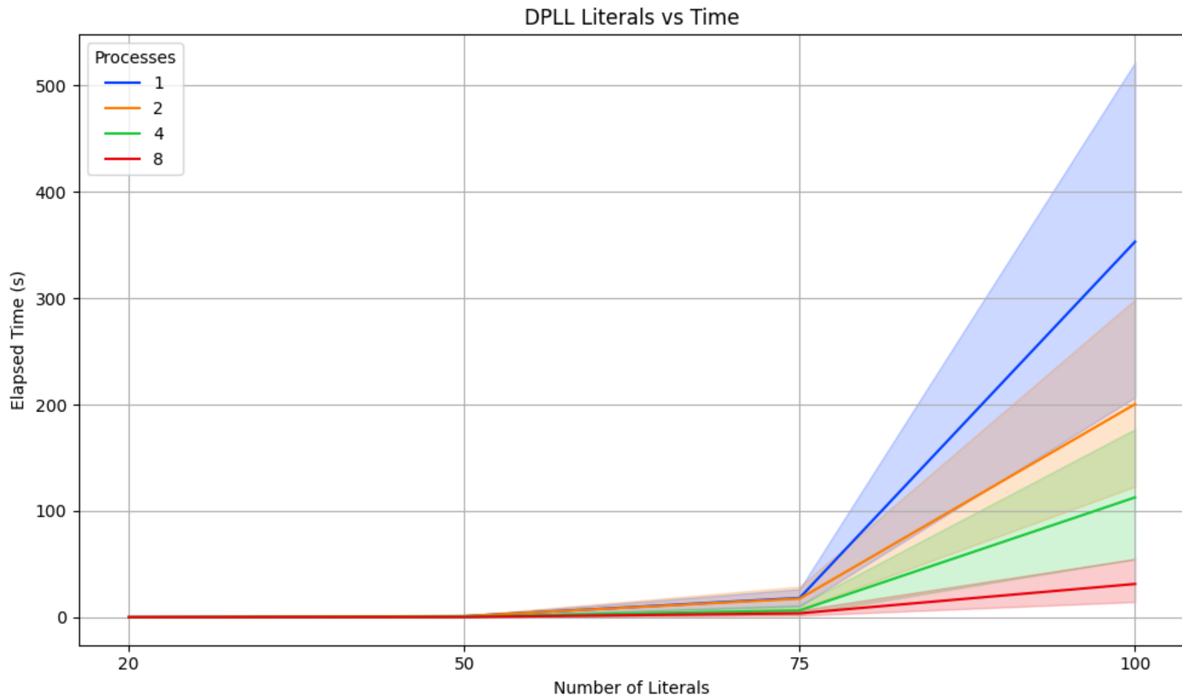

Figure 7: Runtime of DPLL algorithm with 20 to 100 variables (all formulas satisfiable)

Figure 7 shows the runtime of the DPLL algorithm with different numbers of processes (blue = sequential, orange = parallel with two processes, green = parallel with four processes, Red = parallel with eight processes) in seconds for satisfiable formulas. Each formula with 20 literals has 91 clauses, each with 50 literals has 218 clauses, each with 75 literals has 325 clauses, and each with 100 literals has 430 clauses. 25 formulas were used for each of their formula sizes. The line shows the average runtime, while the shadow shows the deviation.

It is not surprising that the average runtime and the amount of deviation both increase with the formulas getting more complex. Increasing the number of processes drastically decreases the runtime and the amount of deviation. This happens for the reason that the likelihood of quickly finding a variable assignment that satisfies the formula increases with each process added. Furthermore, increasing the number of processes also decreases the variable assignment space each process has to go through, reducing the runtime. Another observation we can make is that the runtime is roughly inversely proportional to the number of processes.

5.3.2 UNSAT Cases

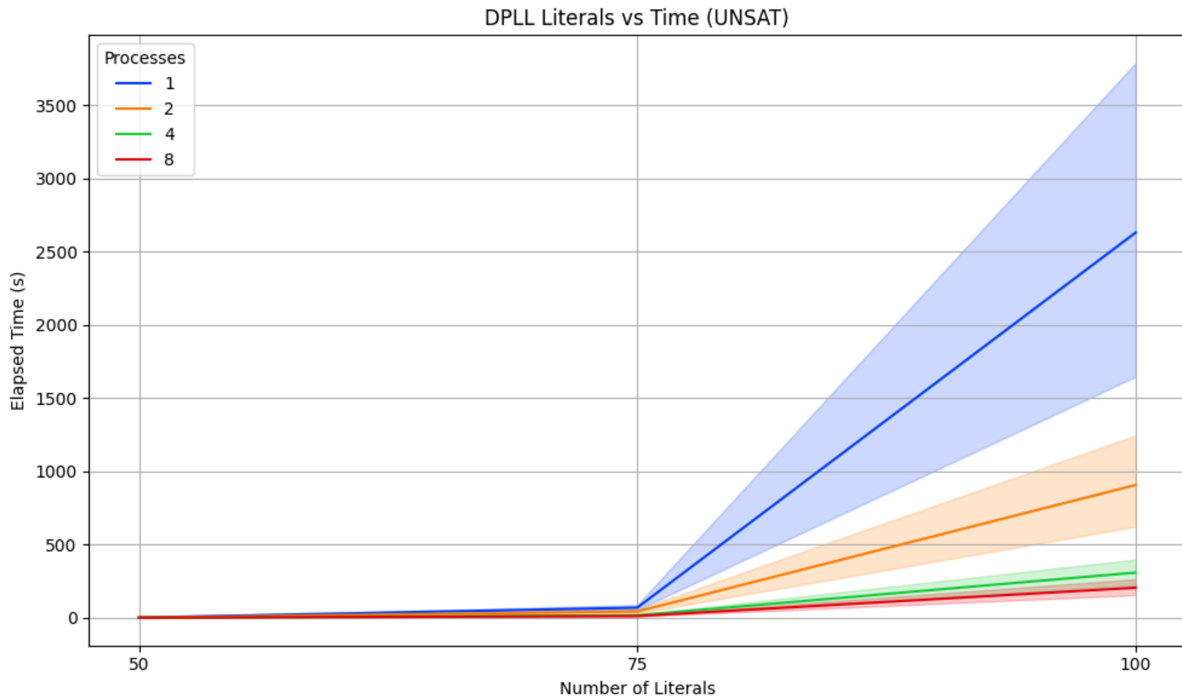

Figure 8: Runtime of DPLL algorithm with 50 to 100 variables (all formulas not satisfiable)

Figure 8 shows the runtime of the DPLL algorithm with different numbers of processes (blue = sequential, orange = parallel with two processes, green = parallel with four processes, Red = parallel with eight processes) in seconds for not satisfiable formulas. The number of clauses and the number of test formulas per number of variables is equal to those of Figure 7, except we did not use formulas with 20 variables for this test.

We can make the same observations as before: The runtime increases with bigger formulas, and the runtime and the deviation both decrease with more processes. Furthermore, we can see that not satisfiable formulas, on average, take more time to solve than satisfiable formulas of the same size. This is because the algorithm has to check more variable assignments when the formula is not satisfiable since, in this case, it has to cover all possible variable assignments. The performance increase of parallelization is even more significant than that of greater than inversely proportional. For example, the average runtime of the sequential (blue) algorithm for formulas with 100 variables is roughly 2600 seconds, while the parallel version with two processes only needed approximately 900 seconds on average. This is about a third of the original runtime, while the number of processes only increased by a factor of 2.

5.4 CDCL Solver

5.4.1 SAT Cases

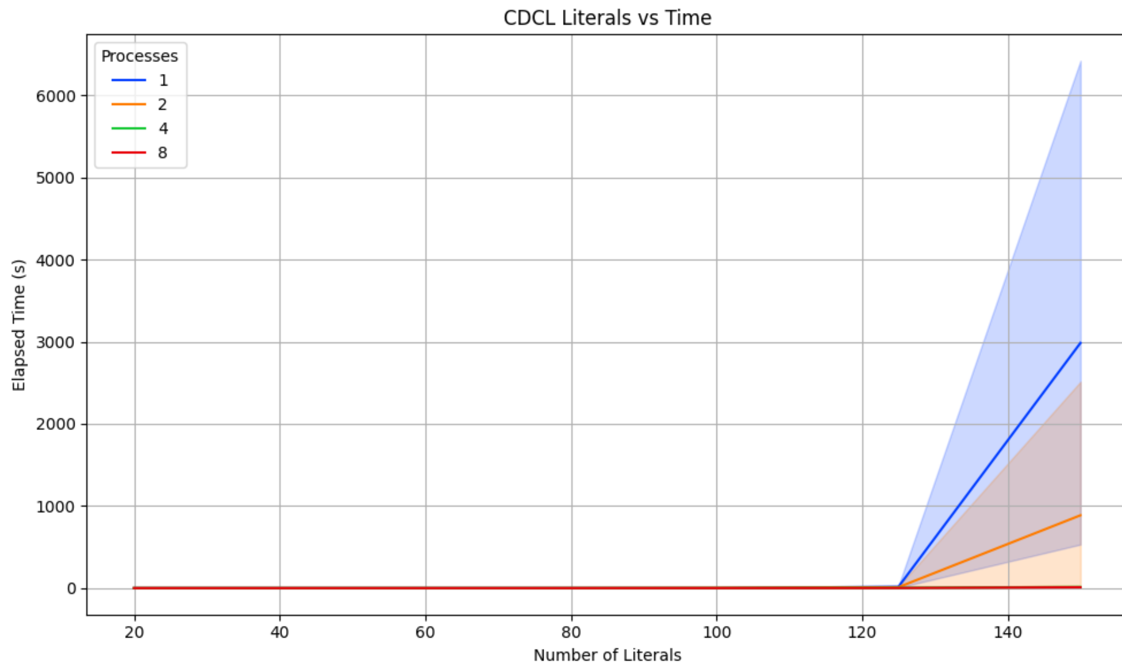

Figure 9: Runtime of CDCL algorithm with 20 to 150 variables (all formulas satisfiable)

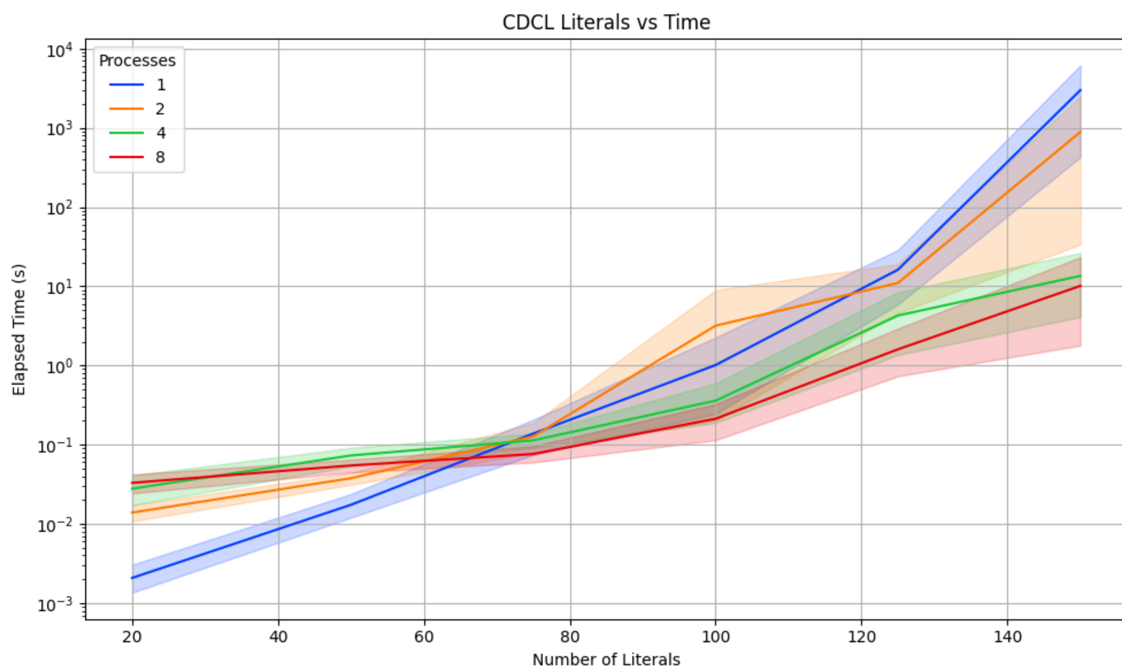

Figure 10: Runtime (in log scale) of CDCL algorithm with 20 to 150 variables (all formulas satisfiable)

In the first set of tests, we evaluated the CDCL solver on satisfiable (SAT) cases, with literals ranging from 20 to 150. We measured the performance of the sequential solver and the parallel versions, which were run with two, four, and eight processes.

Small Instances (20-70 literals): The sequential solver consistently outperformed the parallel versions for problem instances with fewer than 70 literals. This is likely due to the overhead introduced by MPI. Since these smaller instances were solved quickly, the parallel solvers did not have enough work to outweigh the overhead, leading to slower overall performance.

Larger Instances (70-150 literals): Beyond 70 literals, the parallel solvers became increasingly faster than the sequential solver. As the complexity of the SAT instances grew, the benefits of dividing the problem space among multiple processes outweighed the MPI overhead. Notably, the sequential solver took up to 6000 seconds to solve the largest problem with 150 literals, whereas the parallel solvers performed significantly better.

All solvers solved instances with 80 literals or fewer in under seven seconds, but the time difference became substantial for larger problems. As the number of literals approached 150, more processes resulted in faster solving times. The effect of this scaling is best visualized in two graphs: one using a logarithmic scale (Figure 10) to handle the wide range of times and one using a linear scale (Figure 9) to show the time differences in the less complex cases.

5.4.2 UNSAT Cases

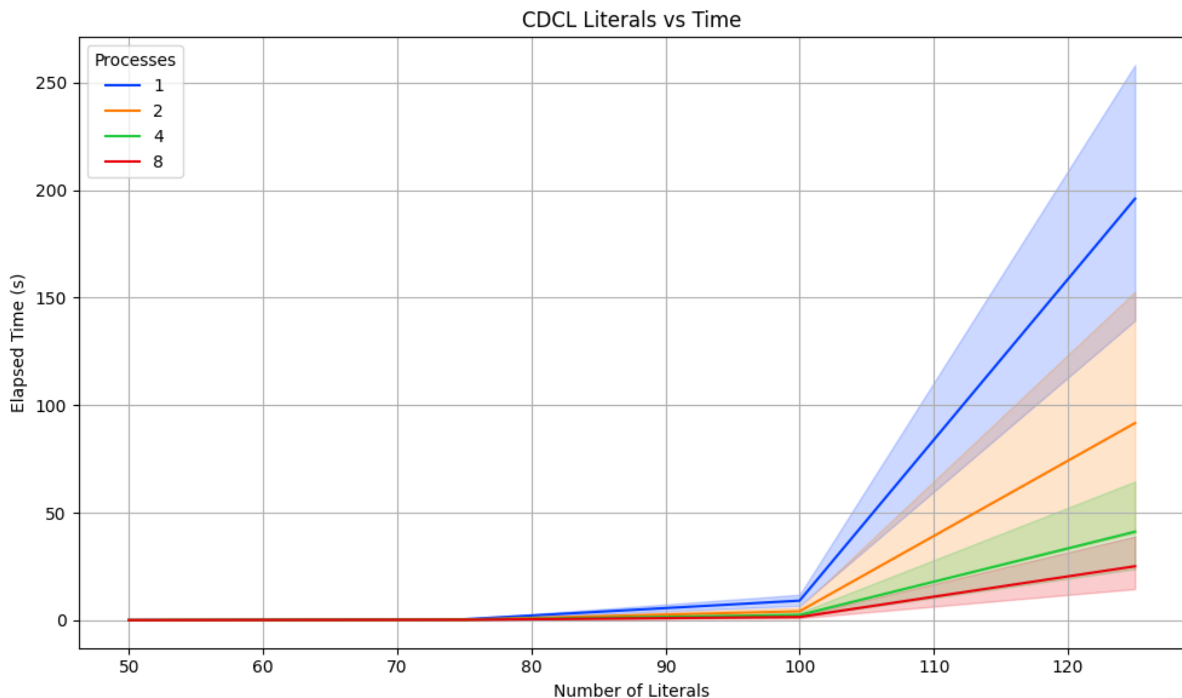

Figure 11: Runtime of CDCL algorithm with 20 to 125 variables (all formulas unsatisfiable)

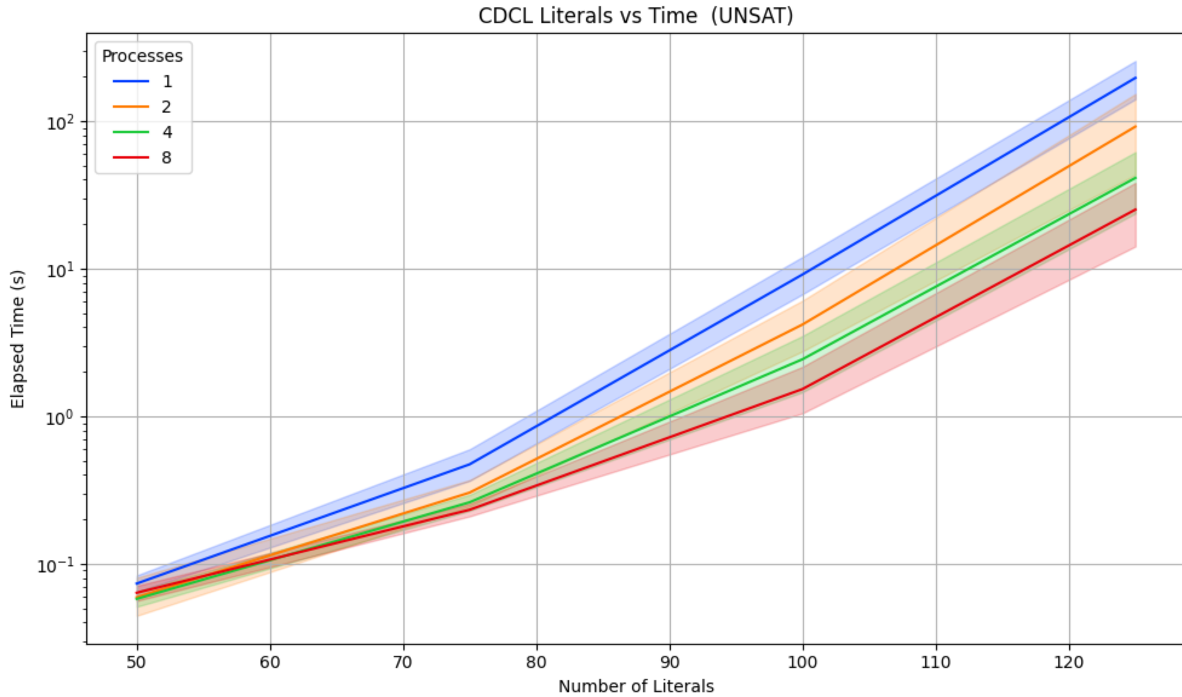

Figure 12: Runtime (in log scale) of CDCL algorithm with 20 to 125 variables (all formulas unsatisfiable)

For the second set of tests, we ran the CDCL solver on unsatisfiable (UNSAT) cases, with a range of literals from 50 to 125. These instances are generally more difficult for SAT solvers, as the entire search space needs to be explored to prove that no solution exists conclusively. The result can be seen in Figure 11 and Figure 12.

Across almost all UNSAT test cases, the parallel solvers were faster than the sequential solver, even for smaller instances. This is interesting because, despite the MPI overhead, the parallel solvers still benefited from multiple processes. This suggests that the search space reduction provided by independent solver decisions in the portfolio-based approach was enough to offset inter-process communication costs.

Unlike the SAT cases, we did not run tests with 150 literals for UNSAT cases, as the sequential solver took too long to complete. This highlights a limitation of our current CDCL implementation, which struggles with large UNSAT instances. Even with parallelization, the solver's performance degrades significantly as the problem size increases for these particularly hard cases.

5.5 Discussion

5.5.1 Brute Force

For the brute force algorithm, we have achieved our goal of reaching a performance increase, which is approximately inversely proportional to the number of processes only for not satisfiable formulas, which is the worst case for the brute force algorithm. The decrease in average runtime for satisfiable formulas is, especially for very simple formulas, barely noticeable. Reaching a decrease in runtime, which would satisfy our objective, is unrealistic for the brute force algorithm because of how it works and is parallelized. For

the runtime to be divided by the number of processes, the algorithm must go through each possible variable assignment. This only happens if the formula is not satisfiable.

5.5.2 DP

Our implementation of the DP algorithm resulted in the opposite of what we planned. The average runtime increases when more processes are added. While this can partly be explained by the overhead added by MPI (especially for very simple formulas), this does not explain the runtime spikes at random variable numbers. Another problem we encountered with the DP algorithm is memory consumption. While testing, the algorithm was aborted multiple times when we used bigger test formulas.

Both problems can be traced to how our implementation chooses a variable for the resolution. The sequential algorithm chooses the first variable of the first clause, while the different processes of the parallel version either take the first variable of the last clause or the first variable of the x-th clause, where x is the respective process rank of the process. This way, both versions of the algorithm will sometimes choose a variable whose resolution will lead to many new clauses, while few or even none can be declared "true" and, therefore, removed. This can cascade into an enormous formula vector (thousands of clauses), leading to memory problems and poor runtimes since the algorithm still has to go through this massive formula vector multiple times per recursion step. This also means that the different processes do not stop immediately when one process finds a solution since each process only checks once per recursion step for a stop signal, explaining the random spikes in runtime. The likelihood of this happening also increases when more processes are added.

To fix this, we have to change the way the algorithms choose variables for resolution. It should depend on the formula, the number of occurrences of each variable, and the size of the clauses a variable occurs in. It is also possible that additional checks for clause satisfiability besides unit propagation, pure literal elimination, and the elimination of clauses containing a variable and its negation can remedy this problem.

5.5.3 DPLL

We have seen that the parallelization of the DPLL algorithm results in significantly faster runtimes when the number of variables exceeds 75. Furthermore, we achieved our objective of the average runtime being approximately inversely proportional to the number of processes for the DPLL algorithm if the formulas are satisfiable. If we only look at not satisfiable formulas, the performance increase is even greater than that. However, the performance increase seems to get smaller as the number of processes increases. Taking the not satisfiable formulas with 100 variables as an example, The average runtime for the sequential algorithm is roughly 2600 seconds. Using the parallel version of the DPLL algorithm with two processes, we reduce the average runtime to approximately 900 seconds. So, we managed to reduce the runtime to about a third by doubling the number of processes. When we double the number of processes again (four processes), we get an average runtime in the neighborhood of 300 seconds. This is, again, 3/4 of the previous runtime. However, if we double the number of processes again (eight processes), we get a runtime of roughly 200 seconds. This is more than a third; therefore, we have reached diminishing returns. The reason for this is probably the overhead added by MPI.

Overall, we achieved our objective of an average runtime for the DPLL algorithm that was approximately inversely proportional to the number of processes.

5.5.4 CDCL

Our tests have shown that the parallel CDCL solver came close to our ideal performance scaling with enough literals in the test cases. In tests with inputs with over 70 literals, the time taken by the parallel solver significantly decreased the more processes we used.

In smaller test samples with less literals, the overhead of MPI's often outweighed the benefits of parallelization. This lead to slower performance compared to the sequential solver. This was more noticeable for instances with fewer than 70 literals, where the parallel solver performed worse than the sequential one.

For UNSAT samples, the parallel solver showed near-ideal scaling across most test cases. This shows its efficiency in exploring multiple search paths in parallel and resolving conflicts more quickly across multiple processes.

5.5.5 Comparison of the different algorithms

In this subsubsection, we will only compare the performance of the brute force, DPLL, and CDCL algorithms since we encountered problems with implementing the DP algorithm.

First, the more advanced algorithms are, on average, faster than the simpler algorithms. The sequential brute force algorithm already needs approximately 34 seconds to solve a not satisfiable formula with 21 variables. On the other hand, the DPLL and CDCL algorithms find a solution for a formula like this in less than a second. We will focus on more complex formulas to compare the performance of the CDCL and DPLL algorithms. When we take a look at satisfiable formulas with 100 variables, for example, the sequential CDCL algorithm only needs roughly one second on average, while the CDCL algorithm needs approximately 350 seconds. The trend of the CDCL algorithm being, on average, the fastest can be observed, no matter how complex the formula is or if it is satisfiable or not.

We can also see that the average runtime of each algorithm is shorter when the formula is satisfiable. If we look again at formulas with 100 variables, the average runtime of the sequential DPLL algorithm is roughly 350 seconds when the formula is satisfiable. The average runtime for not satisfiable formulas of the same complexity is approximately 2600 seconds. The sequential CDCL algorithm has an average runtime of about 1 second for satisfiable formulas with 100 variables, while the average runtime for not satisfiable formulas of similar complexity is a little under 10 seconds.

6 Conclusion

6.1 Summary of Findings

In this project, we developed multiple SAT-solver algorithms. We demonstrated how parallelizing each algorithm can improve runtime performance in SAT problems by implementing and testing sequential and parallel versions. The results of our tests have shown that for smaller input formulas, the overhead of parallelization often outweighs the benefits, causing the sequential solvers to perform faster. However, the parallel solvers perform significantly better as the input size increases. Sometimes, it is even inversely proportional to the number of processes. In the end, with bigger problem sizes, the runtime is inversely proportional to the number of processes. For UNSAT cases, the parallel solvers showed an even more consistent improvement.

In Summary, this project showed the effectiveness of parallelization. Again, the project is open-source and can be seen in our Gitlab Repository.

6.2 Future Work

While all solvers can be further optimized, the CDCL solver will be the most important one. Some improvements can be made:

- Learned Clause Deletion: Learned clause deletion would allow the solver to remove less useful clauses, retaining only the most valuable ones. This could reduce memory usage and improve runtime.
- Clause Sharing: An improvement for the parallel CDCL solver could be the implementation of clause sharing between processes. Sharing learned clauses between solvers may help in pruning the search space.
- Dynamic Process Assignment: Instead of assigning variables based on VSIDS ranking for the parallel solver, implementing dynamic process assignment could allow processes to adaptively change their focus based on the progress of other processes.

References

[24a] *CPU Partitions :: Documentation for HPC*. 2024. URL: https://docs.hpc.gwdg.de/how_to_use/compute_partitions/cpu_partitions/index.html (visited on 09/17/2024).

[24b] *Solvers in Conda*. 2024. URL: https://docs.conda.io/projects/conda/en/latest/dev-guide/deep-dives/solvers.html (visited on 09/17/2024).

[Bec06] Bernhard Beckert. *Vorlesung Logik für Informatiker*. University Lecture, University of Koblenz-Landau. 2006. URL: https://formal.kastel.kit.edu/~beckert/teaching/Logik-SS06/06AussagenlogikResolution.pdf (visited on 09/23/2024).

[BGV99] Randal E. Bryant, Steven M. German, and Miroslav N. Velev. "Microprocessor Verification Using Efficient Decision Procedures for a Logic of Equality with Uninterpreted Functions". In: *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. TABLEAUX '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 1–13. ISBN: 3540660860.

[Bla21] Paul E. Black. *"NP-complete" in Dictionary of Algorithms and Data Structures*. 2021. URL: https://www.nist.gov/dads/HTML/npcomplete.html (visited on 09/17/2024).

[BS97] Roberto J. Bayardo and Robert C. Schrag. "Using CSP look-back techniques to solve real-world SAT instances". In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*. AAAI'97/IAAI'97. Providence, Rhode Island: AAAI Press, 1997, pp. 203–208. ISBN: 0262510952.

[Bur08] John Burkardt. *CNF Files*. 2008. URL: https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html (visited on 09/17/2024).

[CKT91] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. "Where the really hard problems are". In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'91. Sydney, New South Wales, Australia: Morgan Kaufmann Publishers Inc., 1991, pp. 331–337. ISBN: 1558601600.

[Coo71] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[DP60] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *Journal of the ACM* 7.3 (July 1960), pp. 201–215. DOI: 10.1145/321033.321034.

[Hak85] Armin Haken. "The intractability of resolution". In: *Theoretical Computer Science* 39 (1985). Third Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 297–308. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(85)90144-6. URL: https://www.sciencedirect.com/science/article/pii/0304397585901446.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Upper Saddle River, NJ: Pearson, June 2005.

[LSZ97] Michael Luby, Alistair Sinclair, and David Zuckerman. “Optimal speedup of Las Vegas algorithms”. In: *Information Processing Letters* 47 (Apr. 1997), pp. 173–180. DOI: 10.1016/0020-0190(93)90029-9.

[ML62] George Logemann Martin Davis and Donald Loveland. “A machine program for theorem-proving”. In: *Communications of the ACM* 5.7 (July 1962), pp. 394–397. DOI: 10.1145/368273.368557.

[MS] J.P. Marques Silva and K.A. Sakallah. “GRASP-A new search algorithm for satisfiability”. In: *Proceedings of International Conference on Computer Aided Design*. ICCAD-96. IEEE Comput. Soc. Press. DOI: 10.1109/iccad.1996.569607. URL: http://dx.doi.org/10.1109/ICCAD.1996.569607.

[MS99] J.P. Marques-Silva and K.A. Sakallah. “GRASP: a search algorithm for propositional satisfiability”. In: *IEEE Transactions on Computers* 48.5 (May 1999), pp. 506–521. ISSN: 0018-9340. DOI: 10.1109/12.769433. URL: http://dx.doi.org/10.1109/12.769433.

[NSR02] Gi-Joon Nam, K.A. Sakallah, and R.A. Rutenbar. “A new FPGA detailed routing approach via search-based Boolean satisfiability”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.6 (2002), pp. 674–684. DOI: 10.1109/TCAD.2002.1004311.