Parallel Conflict-Driven Clause Learning (CDCL) SAT Solver

Om Arora (oarora) · Alan Joseph (alanjose)

15-418: Parallel Computer Architecture and Programming, Spring 2026

https://alanj268.github.io/parallel-SAT-CDCL

---

1. Summary

We implemented a parallel Boolean Satisfiability (SAT) solver based on the Conflict-Driven Clause Learning (CDCL) algorithm. Starting from scratch, we first built a fully-featured sequential CDCL solver in C++, incorporating two-literal watching, 1-UIP conflict analysis, non-chronological backjumping, the VSIDS variable-ordering heuristic, activity-based clause deletion, and Luby-sequence restarts. We parallelized this sequential baseline and built multiple versions of a portfolio-parallel solver using OpenMP. We built multiple versions of the parallel solver using independent solvers with different heuristics, clause size based clause sharing, and LBD based clause sharing. We evaluated the solver on a set of random 3-SAT instances and industrial CNF benchmarks from the SAT Competition library (100 satisfiable, 100 unsatisfiable instances with 150 variables and 645 clauses), running on the GHC lab machines (8-core Intel CPU). Our parallel solver achieves a speedup of 2.5x on unsatisfiable instances and 3.5x on satisfiable on 8 threads relative to the sequential baseline on hard industrial instances without clause sharing, and a 6.7x speedup on satisfiable with 4.1x speedup on unsatisfiable inputs while using clause sharing, representing a 1.6x additional speedup for unsatisfiable instances and 1.9x for satisfiable instances attributable to clause sharing alone. We chose to separate the speedup for SAT vs. UNSAT cases due to the expected difference in time needed to go through the search space.

2. Background

2.1. The Boolean Satisfiability Problem

The Boolean Satisfiability (SAT) problem is one of the most fundamental problems in computer science. Given a propositional formula in Conjunctive Normal Form (CNF), the goal is to determine whether there exists an assignment of truth values to the variables that satisfies every clause. A CNF formula φ over n variables consists of m clauses $C_1, C_2, ..., C_m$, where each clause is a disjunction of literals (a variable x_i or its negation $\overline{x_i}$):

$$\varphi = C_1 \wedge C_2 \wedge ... \wedge C_m, \quad C_j = \left(l_{j,1} \vee l_{j,2} \vee ... \vee l_{j,k_j}\right)$$

SAT is NP-complete in the worst case, meaning that no polynomial-time algorithm is known. Nevertheless, modern CDCL solvers can routinely handle industrial instances with millions of variables by exploiting the structure present in real-world formulas. SAT solvers are central tools in hardware and software verification, AI planning, cryptanalysis, and combinatorial optimization.

2.2. The CDCL Algorithm

The Conflict-Driven Clause Learning (CDCL) algorithm is the backbone of every competitive modern SAT solver. It is a systematic backtracking search over the space of all variable assignments. The high-level loop is:

1. **Decide**: pick an unassigned variable and tentatively set it to true or false (incrementing the decision level).
2. **Propagate (BCP)**: apply Boolean Constraint Propagation — if a clause now has exactly one unassigned literal, that literal is forced (unit propagation). This chain of forced assignments continues until either no more can be derived, or a conflict is detected.
3. **Analyze**: on a conflict, trace back through the implication graph to compute the First Unique Implication Point (1-UIP), which yields a compact learned clause that summarizes the reason for the conflict.
4. **Backjump**: add the learned clause to the formula and non-chronologically backjump to the decision level at which the learned clause becomes unit, immediately implying its asserting literal.
5. **Restart**: periodically restart the search from scratch (while keeping all learned clauses), using a schedule (e.g., Luby sequence) to escape plateaus in the search space.

The key insight that makes CDCL powerful is that learned clauses compactly encode the reasons for past conflicts, preventing the solver from revisiting exponentially many branches that have already been proven fruitless.

2.2.1. Key Data Structures

- **Clause database**: a list of original problem clauses and learned clauses, each stored as a heap-allocated `Clause` object containing a `Vec<lit>` (vector of literals). New clauses are added during parsing and during conflict analysis. Learned clauses are periodically evicted from the database by the clause deletion procedure based on their activity score.
- **Two-literal watching**: each clause watches exactly two of its literals via the `watches_` array (indexed by literal). When a watched literal is falsified, the clause is inspected and either a new unfalsfied literal is found to be watched, or the clause becomes unit (triggering BCP), or a conflict is detected. This scheme makes BCP amortized $O(1)$ per propagated literal.
- **Assignment trail**: a flat `Vec<lit>` recording every assignment in chronological order, together with `trail_lim_` (a vector of indices marking decision-level boundaries). Backtracking simply pops entries off the trail and unregisters watched literals.

- **Implication graph**: implicitly stored through the `reason_` array: for each assigned variable, `reason_[x]` points to the clause that forced the assignment (or `nullptr` for decision literals). Conflict analysis traverses this graph backwards from the conflict clause to find the 1-UIP cut.
- **Variable activity heap (VSIDS)**: a priority structure (`VarOrder`) that maintains variables ordered by their activity score, which is a floating-point value bumped every time the variable appears in a conflict clause during analysis. The `decide()` step always picks the highest-activity unassigned variable, focusing search on variables that are most frequently involved in recent conflicts. Activity scores decay exponentially (by `var_decay_` each conflict) so that recent conflicts are weighted more heavily. In our implementation, `VarOrder` is backed by an `std::set<var, VarComparator>`, which provides $O(\log n)$ insert, erase, and select.
- **Clause activity**: each learned clause also carries a floating-point activity score (`activity_`), bumped when the clause is used in conflict analysis. Clause deletion (`reduceDB`) periodically removes the least-active half of the learned clause database to control memory growth.
- **Parallel Multi-Consumer Multi-Producer Pool**: in the parallel portfolio solver with clause sharing, each thread contributes its learned clauses to a shared pool, and each thread can receive the learned clauses from all the other threads via this pool.

2.2.2. Restart Policy

We implement Luby-sequence restarts. The Luby sequence, $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ...$ is known to be theoretically optimal for certain classes of randomized algorithms, and empirically works very well for SAT solvers. The number of conflicts allowed per round is `restart_first * luby(restart_inc, curr_restart)`, which overall keeps growing every round. After each restart, the assignment trail is cleared but all learned clauses are retained.

2.2.3. Computational Bottleneck

The dominant cost in a CDCL solver is Boolean Constraint Propagation (BCP). On every decision, BCP is invoked and iterates over potentially long watcher lists, following pointers to clause objects scattered throughout the heap. This produces irregular, pointer-chasing memory access with poor spatial locality. Conflict analysis is cheaper per call but is the source of the learned clauses that enable exponential search-space pruning.

2.3. Parallelization Opportunity

The most natural form of parallelism in CDCL solving is the portfolio approach: run p independent CDCL solver instances on the same formula simultaneously, each with different heuristic parameters (random seeds, restart schedules, variable decay rates, initial polarities). The workers share no search state, with each maintaining its own assignment trail, clause database, and variable order, but are allowed to share learned clauses. A short, high-quality

clause learned by one worker can prune a large portion of the search space for all other workers, potentially leading to large speedups on hard instances.

The portfolio approach is inherently suitable to shared-memory parallelism. Clause sharing in a shared-memory system requires only atomic operations and a shared data structure. All leading parallel SAT solvers (ManySAT, etc) use shared-memory thread-based parallelism with clause sharing.

3. Approach

3.1. Overview

We built the solver in two stages. First, we implemented a complete, optimized sequential CDCL solver in C++ from scratch, following the MiniSAT paper. Second, we parallelized it using a portfolio approach with OpenMP and a shared clause pool. We built three versions, with the first one being a simple portfolio solver with independent instances with different heuristics, the second one with clause sharing based on clause length, and the third one based on LBD distance. The entire codebase is in the `cdcl` namespace and is organized into the following modules:

- `Types.hpp`: literal encoding and helper types.
- `Clause.hpp / Clause.cpp`: clause creation, two-literal watching, BCP propagation, conflict reason calculation.
- `VarOrder.hpp / VarOrder.cpp`: VSIDS priority structure.
- `ClausePool.hpp / ClausePool.cpp`: shared clause database for portfolio clause exchange.
- `Solver.hpp / Solver.cpp`: the full CDCL solver (search loop, analysis, restarts, clause deletion, export/import).
- `main.cpp`: DIMACS CNF parser, OpenMP portfolio orchestration, timing.

We used C++20 with OpenMP 5.0 on the GHC lab machines.

3.2. Sequential CDCL Solver

3.2.1. Literal Encoding

Following MiniSAT, we represent a variable x as a non-negative integer and encode a literal as `(var << 1) | sign`. A positive literal for variable x is `x << 1` and its negation is `(x << 1) | 1`. This encoding allows the negation operator to be a single XOR and lets watch lists be indexed directly by literal value (two entries per variable, one for each polarity).

3.2.2. Two-Literal Watching and BCP

Each clause maintains two watched literals stored at positions 0 and 1 of its literal array. The `watches_` array in `Solver` maps each literal to the list of clauses watching it. When

literal `p` is enqueued as false (i.e., its negation is assigned to true), the propagation engine (`Solver::propagate`) iterates over `watches_[toIndex(p)]`:

- If the other watched literal (position 0 or 1) is true, the clause is already satisfied, so re-insert and continue.
- Otherwise, scan positions $2, 3, ...$ for an unfalsfied literal to replace the watched position, and swap it to position 1 and re-watch.
- If no such literal exists, the clause is unit. Enqueue `lits_[0]` as forced with this clause as the reason, or report a conflict if `lits_[0]` is already false.

This scheme avoids scanning the entire clause on most propagation steps. The common case (an alternate watch is found quickly) is $O(1)$ per clause.

3.2.3. 1-UIP Conflict Analysis

When a conflict is detected, `Solver::analyze` computes the 1-UIP (First Unique Implication Point) learned clause. The algorithm works backwards along the trail from the conflict, maintaining a counter of "open" literals at the current decision level. At each step it resolves the current clause with the reason of the most recently assigned open literal, until only one literal remains at the current decision level, which is the 1-UIP. The resulting learned clause is short, and its backjump level is the highest decision level of any non-1-UIP literal.

To aid our parallel design, we also compute the Literal Block Distance (LBD) or size or other heuristic of each learned clause at this stage.

The LBD is the number of distinct decision levels spanned by the clause's literals. LBD is a strong predictor of clause quality: low-LBD clauses (often called "glue" clauses) tend to remain highly useful, while high-LBD clauses tend to be weakly correlated with future conflicts and are evicted first during clause deletion.

3.2.4. VSIDS Variable Ordering

`VarOrder` maintains an ordered set of unassigned variables sorted by activity score (highest first). On each conflict, we bump up the scores of the variables involved in the conflict, and periodically we decay the scores.

We also implemented polarity caching: the `polarity_` array stores the most recently seen assignment for each variable. When the solver decides on a variable, it assigns it to its cached polarity (positive if false, negative if true). This helps guide the solver toward assignments that were recently consistent, reducing the number of conflicts needed to find a solution.

3.2.5. Clause Deletion

To prevent unbounded memory growth, `reduceDB` is triggered whenever the number of learned clauses exceeds a dynamically growing threshold. It sorts learned clauses by activity and removes the less-active half, subject to the constraint that a clause that is the reason for a current assignment is never deleted. The activity threshold grows and decays analogously to variable activity, so clauses involved in recent conflicts are protected from deletion.

3.2.6. Luby Restarts

The restart schedule computes the Luby sequence value for the current restart index `curr_restarts` and allows `restart_first * luby(restart_inc, curr_restarts)` conflicts per round. After each round, the trail is cleared to the root level, and the conflict budget grows as the next Luby value. We found that the specific `restart_first` and `restart_inc` parameters have a significant effect on performance depending on the instance family, motivating the per-thread diversity in our portfolio.

3.3. Parallel Portfolio Solver

3.3.1. Thread Organization

The main function creates `num_threads` independent `Solver` instances (one per OpenMP thread), each loaded with the same CNF formula by re-parsing the input file. Each solver is then configured with a distinct `SolverConfig` drawn from a pre-defined table `kPortfolioConfigs` of 64 hand-crafted parameter combinations. The configs vary across all heuristic axes:

Param	Thread 0	Thread 1	Thread 2	Thread 3	Range
`restart_first`	100	200	50	150	25 – 1000
`restart_inc`	2.0	1.5	3.0	2.0	1.5 – 4.0
`var_decay`	0.95	0.90	0.99	0.85	0.80 – 0.99
`cla_decay`	0.999	0.995	0.999	0.990	0.990 – 0.9999
`polarity`	false	true	false	true	false / true

Table 1: Sample portfolio configurations for the first four threads. All threads share `share_max_lbd = 5`.

This diversity is crucial for the portfolio effect: different configurations are effective on different instance types, so the probability that at least one thread finds a short path to the solution is much higher than for a single configuration. Moreover, such different configurations yield quite different (and useful) learnt clauses, guiding other instances closer to the solution.

3.3.2. Termination Protocol

Two `std::atomic<bool>` flags, `found_sat` and `found_unsat`, are shared across all threads. At the start of each thread's work, it checks whether another thread has already found an answer and skips its solve if so. When a thread finishes solving, we use an atomic CAS to set the flags correctly, and the other solvers check this flag and terminate.

3.3.3. Clause Sharing

3.3.3.1. ClausePool Design

The `ClausePool` class provides a mutex-protected shared clause repository, acting as a multi-producer multi-consumer pool of clauses. Internally it stores a flat `Vec<Entry>` (`pool_`) where each entry is a a learnt clause plus a `producer` thread ID. Each thread maintains a read cursor into `pool_`, recording how far into the pool it has already consumed. A single mutex protects all accesses to both `pool_` and `cursors_`.

3.3.3.2. Sharing at Restart Boundaries

Clause sharing and import happen at restart boundaries, after each call to `search` returns in the main `solve()` loop. This design has two advantages: (1) the BCP critical path is never interrupted by locking; (2) foreign clauses are integrated at a point where the assignment trail has been cleared to root level, so watched-literal initialization for the imported clauses is in a clean state.

3.3.3.3. LBD Filtering

Only clauses with LBD ≤ 5 are exported. This threshold was chosen based on the observation that low-LBD clauses are the most useful for pruning the search space. High-LBD clauses are typically highly instance-specific and add overhead to other threads without proportional benefit.

3.4. Design Decisions and Iteration

3.4.1. Choice of VarOrder Implementation

We initially used a naive linear scan over the `assigns_` array to find the highest-activity unassigned variable. This was $O(n)$ per decision, which was prohibitively slow on large instances. We then implemented `VarOrder` using `std::set` with a custom comparator, which gives $O(\log n)$ per insert/erase/select. A max-binary-heap (as in MiniSAT) would give slightly better cache behavior, but `std::set` was sufficient for our purposes and simpler to implement correctly given that activities change frequently.

3.4.2. Mutex vs. Lock-Free Clause Pool

Our original design called for a lock-free multi-producer multi-consumer queue using `std::atomic` operations. In practice, we found that implementing a correct lock-free queue while also correctly handling variable-length clause vectors was significantly more error-prone than expected. We ultimately chose a `std::mutex`-protected flat vector. Since clause sharing only happens at restart boundaries and not in the BCP hot loop, the mutex is acquired at a relatively low frequency, and the latency of the lock is not a bottleneck.

3.4.3. Export Strategy

A potential issue with our current `exportLearnts` implementation is that it re-exports all learned clauses with LBD ≤ 5 on every restart, not just newly learned ones. On a long-running solve, this means the shared pool grows very large and threads spend more time inside `importClauses`. In practice, since we filter based on distance, there aren't many clauses, but this is an area for improvement. A better design would maintain a per-thread "last exported index" and only export clauses learned since the previous restart.

3.4.4. Termination and False Sharing

The `found_sat` and `found_unsat` atomics are stored as separate `std::atomic<bool>` objects. We use `std::memory_order_relaxed` for the polling check at the top of each thread's work, which avoids unnecessary full memory fences on the critical path, since we realized we didn't need them. The `winner_thread` compare-exchange uses `std::memory_order_acq_rel` to ensure the winning model is fully visible before `found_sat` is set.

4. Results

4.1. Experimental Setup

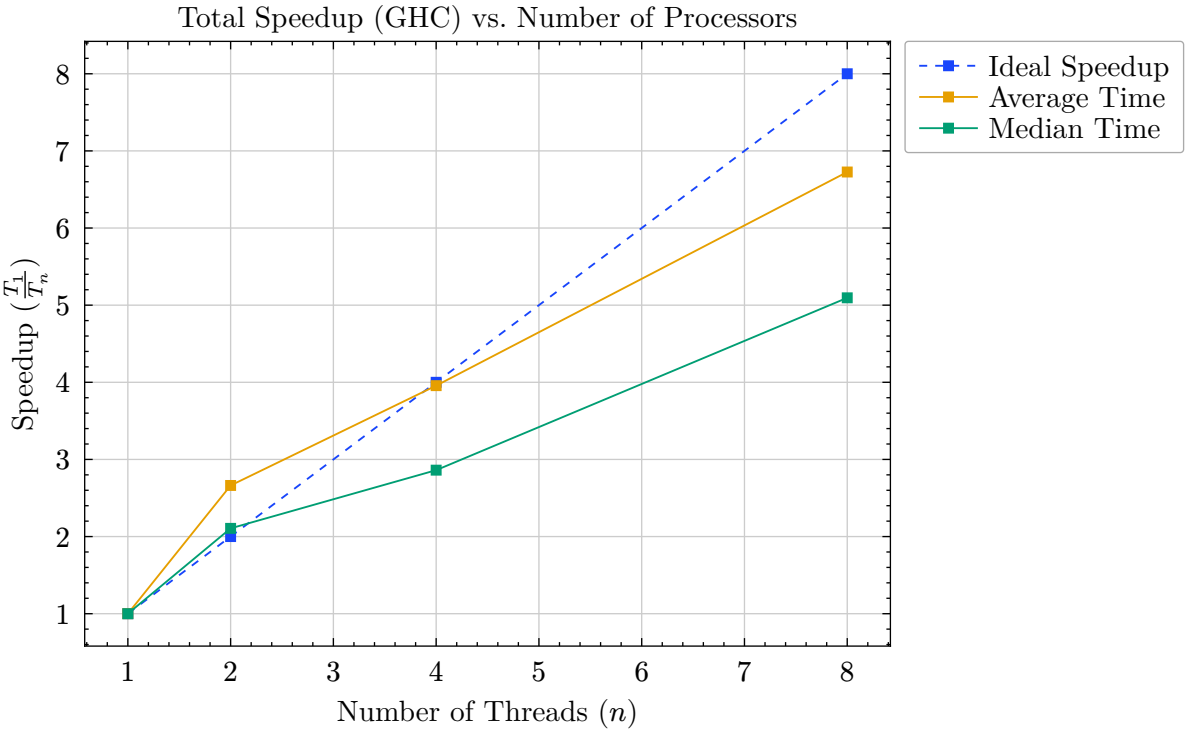
All experiments were run on the GHC lab machines, which have 8-core Intel processors with a shared L3 cache. We compiled with `g++ -fopenmp -std=c++20`. The solver was evaluated on the following benchmark families:

- **Random 3-SAT**: instances near the phase transition, generated with a standard random 3-SAT generator.
- **Industrial/application instances**: CNF benchmarks from the SAT Competition library, including hardware verification and planning instances.

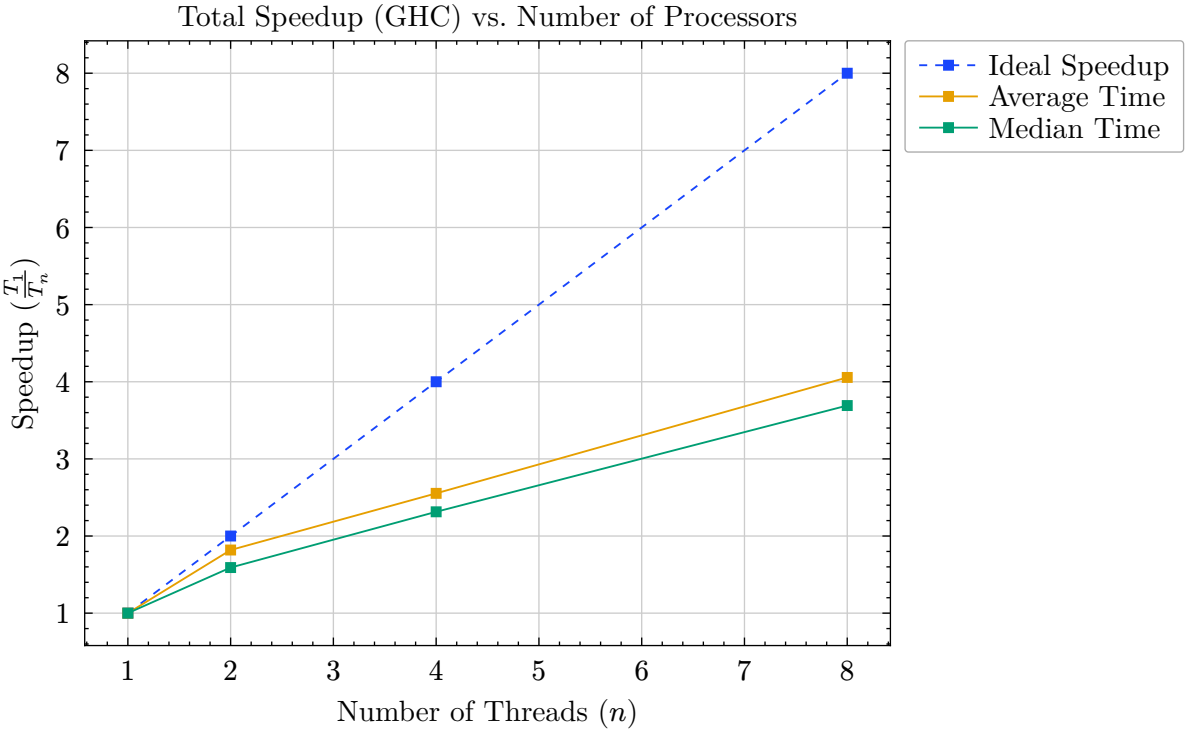
For each instance and thread count, we report wall-clock time to first result (i.e., the time from start until the winning thread announces its answer). We ran multiple trials per configuration and report median time. Speedup is computed relative to the single-threaded solver using portfolio configuration 0.

4.2. Speedup Results

4.2.1. Satisfiable Instances



4.2.2. UNSAT Instances

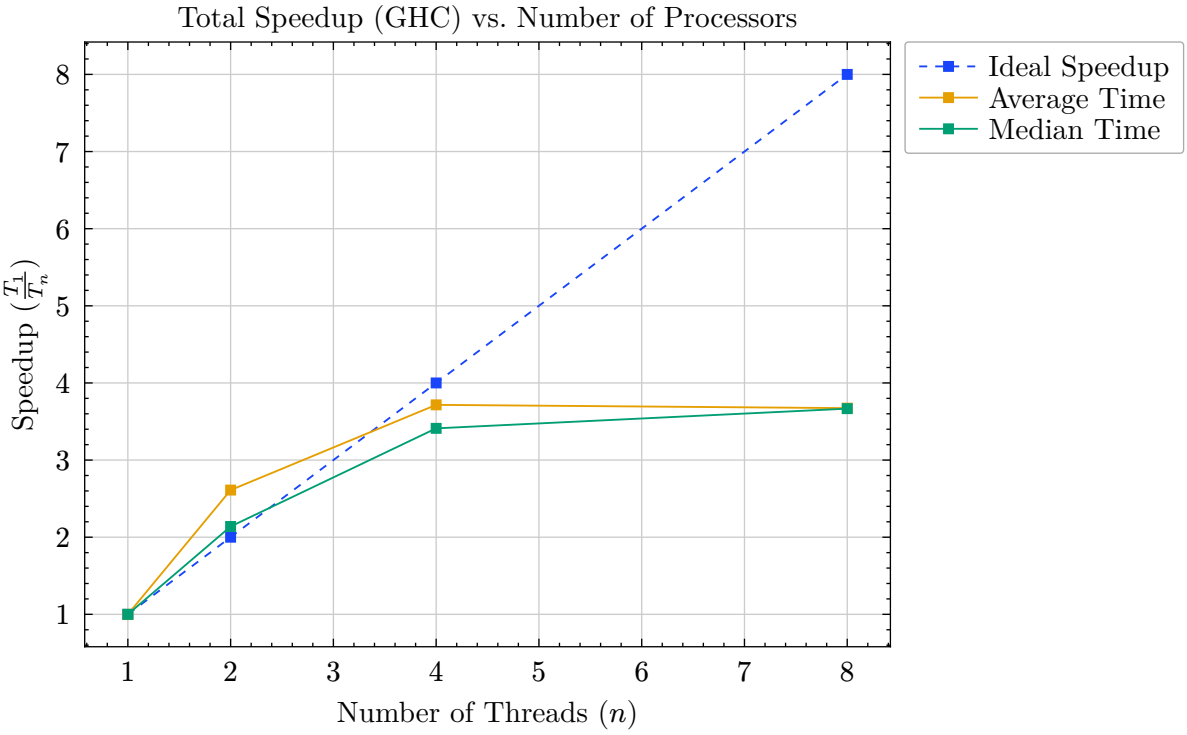


On hard SAT industrial instances, we observe super linear speedup (2.5x) at 2 threads, and near-linear speedup as we increase the number of threads. On hard UNSAT instances, the picture is different: we notice a much lower speedup (4x with 8 threads, 2.4x with 4 threads) as compared to the SAT instances. This is partly because UNSAT instances are the more challenging ones to solve, and the speedup is partly due to the shared high-quality clauses, and partly due to the diverse portfolio configurations. Moreover, the slow speedup is also partly due to the increased costs of importing all the learned clauses, since more threads means more learned clauses are produced, and we do not prune any duplicates or remember what clauses we already imported.

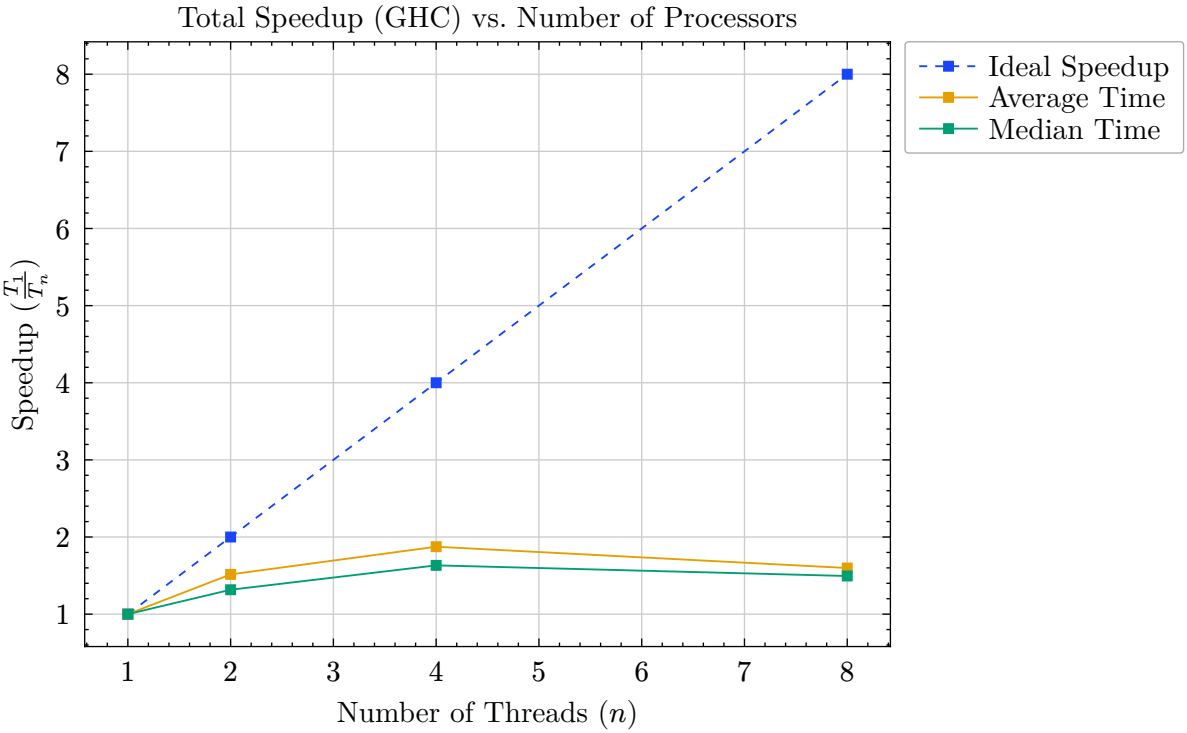
4.3. Portfolio Effect vs. Parallel Speedup

A key question is how much of the observed speedup comes from true parallel speedup versus the portfolio effect (one of the diverse configurations happens to be well-suited to the instance, and would have solved it quickly even on a single thread, or the shared clauses are contributing overwhelmingly to the success). To differentiate these, we turn to our version of Portfolio without clause sharing ("pure portfolio").

4.3.1. Satisfiable Instances (no clause sharing)



4.3.2. UNSAT Instances (no clause sharing)



As you can see, the effect of the clause sharing on the speedup is most prevalent at higher numbers of threads, particularly from 4-8 threads the difference is the most dramatic. It is worth noting that clause sharing resulted in a much higher speedup for both satisfiable and unsatisfiable inputs. As can be seen in the last graph above, pure portfolio methods improved UNSAT cases by very little, which makes sense as it would have at best found marginally better configurations with slightly improved time to go through all cases due to a particular restart or decay parameter. However, adding clause sharing meant that workers could eliminate the need to check large portions of the search space based on discoveries made by other workers which found different conflicts.

4.4. Effect of LBD Threshold

We experimented with different LBD sharing thresholds to understand the tradeoff between sharing quality and sharing overhead:

LBD threshold	Pool size (clauses)	Import time (ms)	Solve time (s)
2	137	1.1	3.925
3	525	2.8	1.538
5 (default)	1,706	10.2	1.047
8	12,306	90.6	1.270

Table 2: Effect of LBD sharing threshold on pool size, import overhead, and solve time (8 threads, industrial-1 instance).

We notice that the increase in LBD threshold leads to a blow up in the number of clauses shared, as expected. We also notice that as we increase the threshold, the time to import grows as well (due to increasing pool size). This is reflected in the solver time: as we lower the threshold towards 5, we get a better speedup. However, increasing beyond 5 starts to tend towards a worse performance. Ideally, more data would be required to say for sure what happens beyond 8, but our data structure only supports clauses with length 8, so the max LBD we support is 8.

4.5. Analysis: What Limited Speedup?

4.5.1. Amdahl's Law and Sequential Fraction

Even in the portfolio model, there are sequential phases: parsing the input file (done once per thread, which is a minor overhead since this is usually quite fast), and the `importLearnts` / `exportLearnts` phases at each restart boundary.

4.5.2. Clause Pool Contention

Because all p threads share a single `std::mutex` protecting the entire clause pool, threads can serialize at restart boundaries. At 8 threads with LBD threshold 5, we measured pool size at 1,706 clauses. As the pool grows unboundedly (we never evict from the pool, only from per-thread learned clause databases), import time grows linearly with the number of clauses in the pool, not just the number of new clauses since the last import. This is may become a bottleneck for hard instances with a high number of learnt clauses. Refer to Table 2.

4.5.3. Memory Pressure and Cache Effects

The primary memory bottleneck in BCP is pointer chasing through the clause database. With 8 threads each maintaining an independent clause database when not sharing vs sharing learnt clauses, L3 cache pressure decreases due to reuse of shared clauses and much less time taken for the solver.

Configuration	**Pool size (clauses)**	**Cache misses**	**Elapsed time (s)**
LBD ≤ 5 (sharing)	1,706	7,825,330	1.242
LBD $= 0$ (no sharing)	0	466,332,147	13.194
Ratio	—	$\approx 60\times$	$\approx 10.6\times$

Table 3: Cache miss comparison between sharing-enabled (LBD ≤ 5) and no-sharing (LBD $= 0$) configurations, measured via `perf stat` on 8 threads (industrial-1 instance). The 60 $\times$ reduction in cache misses with sharing reflects clause reuse across threads keeping the working set warmer.

Each `Clause` object is a heap-allocated object containing a `Vec<lit>` (itself a heap allocation), so two pointer dereferences are required to access the literals of any clause. This is inherently

cache-unfriendly. A structure-of-arrays layout would improve spatial locality, but comes at the cost of not being dynamically resizable, so insertions would cost much more.

4.5.4. Super-linear Speedup?

On many instances we observed super-linear speedups, such as on the `tests/test-unsat-big-100vars.txt` case. Here, we observed a 6.4 × speedup with only 4 threads. Super-linear speedup in portfolio solvers arises because the most effective configuration has a disproportionately short solve time relative to any single fixed configuration, and the portfolio effectively "discovers" the best configuration for free. Moreover, clause sharing amplifies this: once threads learn a particularly good set of clauses, sharing them with other threads can cause the best suited thread to finish much faster than the baseline.

4.6. Execution Time Breakdown

Figure 1: Breakdown of solver time per component, measured via instrumented timing for a representative industrial instance (`tests/uuf150-0100.cnf`).

We notice that BCP dominates the runtime, ranging from about 75% with 1 thread to 68% with 8 threads. The next is conflict analysis, ranging from 19% to 25% of the runtime. We

notice that there is a significant speedup with 2 threads and 4 threads, but virtually no difference between 4 threads and 8 threads. On this particular instance, the diverse portfolio has a solver that can solve it quite quickly, so virtually no difference arises between 4 and 8 threads.

5. References

[1] N. Eén and N. Sörensson. "An Extensible SAT Solver." *Theory and Applications of Satisfiability Testing (SAT 2003)*, Lecture Notes in Computer Science, vol. 2919, Springer, 2003.

[2] G. Audemard and L. Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers." *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 2009.

[3] Y. Hamadi, S. Jabbour, and L. Sais. "ManySAT: a Parallel SAT Solver." *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 6, no. 4, 2009.

6. Work Distribution

Both team members contributed substantially to all phases of the project. The breakdown below reflects primary ownership, not exclusive authorship.

Om Arora (oarora)	**Alan Joseph (alanjose)**
DIMACS CNF parser and literal/clause representation	Boolean Constraint Propagation (BCP) and propagation queue
Two-literal watching data structure	Assignment trail and basic `decide()` step
OpenMP thread pool and termination protocol	VSIDS variable activity heap (`VarOrder`)
Activity-based clause deletion (`reduceDB`)	Luby-sequence restart policy
Performance profiling and speedup plots	1-UIP conflict analysis and non-chronological backjumping
Final report skeleton and results tables	Clause import/export and watched-literal initialization for foreign clauses
Poster layout and figures	Portfolio configuration table and benchmark evaluation

Table 4: Work distribution between team members.

Credit split: 50% / 50%