Formal Methods in Computer-Aided Design 2026

Parallelizing Congruence Closure

Zachary Kent*, Amar Shah*
Carnegie Mellon University, Pittsburgh, PA, USA
{zkent, amarshah}@cs.cmu.edu

***Abstract*—Given a set of ground equalities over terms with uninterpreted function symbols, congruence closure computes the smallest equivalence relation containing these equalities that respects functional congruence. While congruence closure is a performance-critical component of SAT solvers, satisfiability modulo theories (SMT) solvers, and equality saturation engines, all existing implementations are sequential. In this work, we provide the first parallel algorithms for congruence closure.**

We build on two ideas from the parallel algorithms literature: a union-find data structure that supports concurrent union operations and a semisort (group-by) algorithm that allows us to discover new congruences in parallel. We present two algorithms that replace the classic sequential worklist with a bulk-synchronous closure loop: each round processes all pending unions in parallel, detects new congruences, and repeats until a fixpoint. The two algorithms differ in how they identify which terms to re-examine in each round. We evaluate both these algorithms against a sequential baseline on a number of benchmarks and show near-linear speedup on up to 32 cores.

I. Introduction

Congruence closure is an important algorithm for satisfiability modulo theories (SMT) solving [1], equality saturation [2], type unification [3], and hardware verification [4]. Congruence closure determines whether a set of equalities and disequalities between terms is satisfiable. For example, given the equalities $a = b$ and $b = c$ and the disequality $f(a) \neq f(c)$, congruence closure will conclude that this is unsatisfiable: $a = c$ follows by *transitivity*, and then $f(a) = f(c)$ follows by *congruence*, contradicting the disequality. To decide this problem in practice, one must maintain equalities using a union-find data structure and propagate congruences to parent terms.

Congruence closure is a component of SAT solvers like Kissat [5]; equality saturation engines like egg [6] and egglog [7]; and SMT solvers like Z3 [8] and cvc5 [9]. These tools are used in a variety of applications, including circuit equivalence checking [4], software verification [10, 11], hardware design [12], and mathematical theorem proving [13]. Hence, their performance is critical, and much work has gone into improving their efficiency. One promising avenue for improving the performance is *parallelization*: utilizing multiple processors to solve the problem faster.

To this end, we present two novel parallel congruence closure algorithms. Our algorithms use a concurrent union-find data structure and a bulk-synchronous parallel design to efficiently compute the congruence closure of a set of equalities and disequalities.

At a high level, our algorithms replace the classical sequential worklist of Nelson and Oppen [14] with a *bulk-synchronous* closure loop. Initial equalities are unioned into a lock-free concurrent union-find [15]. The algorithm then proceeds in rounds: each round identifies a frontier of terms whose congruence class may have changed, discovers all new congruences among them in parallel, and unions the resulting classes. We implement each round using standard data-parallel primitives provided by ParlayLib [16]. The algorithm terminates when no round discovers a new merge, exactly mirroring the fixed point of the sequential procedure.

The two algorithms differ in how they detect new congruences. The algorithm ParentCC maintains a parent list for each congruence class, and when classes are merged, the parents of the merged classes are re-examined in the next round. The second algorithm FilterCC marks a class as dirty when it is merged and then filters all terms to find those with a child in a dirty class.

In summary, this paper makes the following contributions:

- We present two parallel congruence closure algorithms, ParentCC and FilterCC, both built on a bulk-synchronous parallel design that replaces the classical sequential worklist with rounds of data-parallel primitives over a lock-free concurrent union-find.
- We characterize when our algorithms achieve good speedup using the notions of congruence *depth* and *width*, and show empirically that, on our benchmarks, width is the dominant factor.
- Despite congruence closure being P-complete (i.e. believed to not have a theoretically efficient parallel algorithm) [3], we demonstrate near-linear speedups up to 32 cores over an efficient sequential baseline on random benchmarks, a synthetic benchmark suite designed to stress-test our algorithms, and a set of circuit equivalence benchmarks [4].
- We open source our code and benchmarks[1].

II. Background

A. Congruence Closure

Congruence ($\equiv$) requires that two terms with the same function and pairwise-congruent children must be congruent,

*Both authors contributed equally to this research.

[1]Our code is available at https://github.com/amarshah10/ParallelEgraph

https://doi.org/10.34727/2026/isbn.978-3-85448-093-8_25

i.e. $f(s_1, \ldots, s_k) \equiv f(t_1, \ldots, t_k)$ whenever $s_i \equiv t_i$ for every i. *Congruence closure* refers to the problem of building the smallest equivalence relation that contains a given set of equalities and is closed under congruence.

Typically, congruence closure is implemented using a data structure called an *e-graph* [17], which extends a union-find data structure [18] to represent equivalence classes of terms, while maintaining congruence.

Definition 1 (Terms)**.** We define $\mathcal{T}$ to be a set of *terms* over function symbols F, where each $t \in \mathcal{T}$ is either a *leaf term*, for example x, y, z or a *compound term* of the form $f(t_1, \ldots, t_k)$, where $f \in F$ with arity$(f) = k$ and $t_1, \ldots, t_k \in \mathcal{T}$.

For a compound term $f(x)$, we say that x is a *child* of $f(x)$, and conversely that $f(x)$ is a *parent* of x.

Definition 2 (Congruence Closure)**.** Given $(F, \mathcal{T}, E)$ where F is a list of function symbols, $\mathcal{T}$ is a set of terms over F, and $E = \{a_1 = b_1, \ldots, a_n = b_n\}$ is a set of equalities between terms in $\mathcal{T}$ where $a_1, \ldots, a_n, b_1, \ldots, b_n \in \mathcal{T}$, the congruence closure ($\equiv$) of $(F, \mathcal{T}, E)$ is the smallest equivalence closure of $=$ over $\mathcal{T}$ that respects congruence. More specifically:

1) $a_i \equiv b_i$ for every i (input equalities $=$).
2) $s \equiv s$ for every term s (reflexivity).
3) If $s \equiv t$ then $t \equiv s$ (symmetry).
4) If $s \equiv t$ and $t \equiv u$ then $s \equiv u$ (transitivity).
5) If $s_i \equiv t_i$ for every i then $f(s_1, \ldots, s_k) \equiv f(t_1, \ldots, t_k)$ (congruence).

B. Motivating Example

To make the structure concrete, we work through a small instance of a problem where congruence closure shines: *combinational equivalence checking* for hardware circuits [4]. Given two circuits C_1 and C_2 with the same inputs and outputs, one can check their equivalence using a construction known as a *miter*: a single circuit that feeds the shared inputs to both copies, XORs their corresponding outputs, and ORs the results. The miter outputs 1 on some input iff C_1 and C_2 disagree on that input, so C_1 and C_2 are equivalent iff the formula "the miter outputs 1" is unsatisfiable.

Figure 1 shows a simple miter, adapted from the running example of Biere et al. [4], with two copies of the same three-gate circuit. Each side $i \in \{1, 2\}$ computes

$$a_i = \mathrm{AND}(r, s),$$
$$x_i = \mathrm{XOR}(u, v),$$
$$m_i = \mathrm{ITE}(c,\ a_i,\ x_i),$$

where r, s, u, v, c are shared inputs but the two sides use *distinct* gate instances (a_1, a_2 are different gate-IDs, even though both compute $\mathrm{AND}(r, s)$, and similarly for x_i and m_i). The two outputs m_1, m_2 are combined by a top-level $\mathrm{XOR}(m_1, m_2)$, and the miter asserts $p = m_1 \oplus m_2 = 1$.

Miters are typically encoded into conjunctive normal form (CNF) and handed to a SAT solver, but these formulas are large and the CNF encoding destroys the gate-level structure that makes equivalence apparent. Biere et al. [4] observed

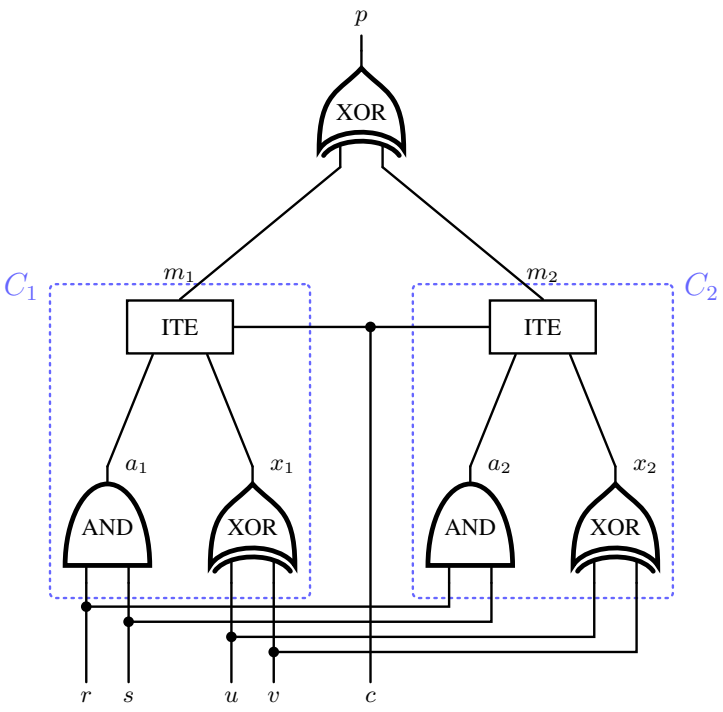


Fig. 1: Miter for combinational equivalence checking, adapted from Biere et al. [4]. Two structurally identical sub-circuits C_1 and C_2 (dotted boxes) share inputs r, s, u, v, c. Each side computes $a_i = \mathrm{AND}(r, s)$, $x_i = \mathrm{XOR}(u, v)$, and $m_i = \mathrm{ITE}(c, a_i, x_i)$. The top XOR asserts $p = m_1 \oplus m_2 = 1$; the two sub-circuits are equivalent iff this formula is unsatisfiable.

that miters generated from real designs often contain many syntactically distinct gates that compute the same function, so running congruence closure over the recovered gates can drastically simplify (and often immediately decide) the resulting formula.

On this miter, congruence closure proves equivalence in three rounds, the first of which discovers two *independent* congruences in parallel:

1) **Round 1 (leaf gates).** Both AND-gates have the same function symbol and the same children (r, s), so $a_1 \equiv a_2$ by congruence. Likewise both XOR-gates compute over (u, v), giving $x_1 \equiv x_2$.
2) **Round 2 (ITE gates).** With $a_1 \equiv a_2$ and $x_1 \equiv x_2$ established, the two ITE-gates now have pointwise-congruent children (c, a_i, x_i), so $m_1 \equiv m_2$.
3) **Round 3 (top XOR).** The top XOR's two children are now in the same equivalence class, so by congruence its output equals $\mathrm{XOR}(m_1, m_1)$. If this formula is given to a SAT solver, the solver can immediately determine that the output is 0 and thus the two circuits are equivalent.

Two features of this trace are important for our work. First, the congruences at each round are *mutually independent*: $a_1 \equiv a_2$ and $x_1 \equiv x_2$ do not depend on each other and could be discovered concurrently, as could the merges at each subsequent

level. This is exactly the parallelism our algorithms exploit. Second, the trace naturally separates into *depth* (the number of rounds) and *width* (the number of new congruences per round). In this tiny example both are small, but in real benchmarks the width can grow to thousands or millions of independent merges per round. We formalize these notions next.

Definition 3 (Congruence Depth)**.** We say that two terms are k-step congruent if they are congruent after k rounds of congruence closure. The *congruence depth* D of a formula is the smallest k such that every pair of congruent terms in the formula is k-step congruent.

Definition 4 (Congruence Width)**.** For each round r of congruence closure, let *width*(r) denote the number of new equalities added in round r. The *congruence width* of a problem is $\max_{r \in 0 \ldots D}$ *width*(r), i.e. the maximum number of new equalities added in a single round.

In general, if we have a large number of terms that become congruent in a single round, we can add all of these terms to the union-find concurrently. This motivates our parallel algorithms for congruence closure, which process each round of congruence closure in parallel.

C. Parallel Algorithms

We describe our algorithms in the standard *work–span* model of shared-memory parallelism [19]. The *work* W of a computation is the total number of operations it performs; the *span* (or depth) S is the length of its longest sequential dependency chain. A famous result [20, 21] states that on p processors with a greedy scheduler, the running time is bounded by $O(\max(W/p, S))$, so an algorithm exposes useful parallelism when W/S, its *parallelism*, is large compared to the available cores.

However, congruence closure is known to be P-complete [3], i.e. it can be solved in polynomial time (P) and every problem in P can be reduced to it using a logarithmic-space reduction.

P-complete problems are widely believed not to be solvable in poly-logarithmic span and polynomial work, so they are not efficiently parallelizable. However, this does not preclude the existence of a parallel algorithm that works well on most practical examples. We discuss such an algorithm in the next section.

For this paper, we assume an arbitrary-forking Multi-Process Random-Access Machine (MP-RAM) model of parallelism [19]. In this model, we have a number of processors sharing an unbounded memory. Each processor performs sequential computations and is extended with a FORK instruction that can spawn k new threads.

In practice, we will describe our algorithms using standard parallel constructs implemented in the PARLAYLIB library [16], which provides a high-level interface for parallel programming in C++. For instance, we will use a parallel for loop (**parfor**) to express the parallel processing of a set of independent tasks. It is easy to see how this could have been implemented using FORK, where k is the number of iterations in the loop.

Concurrent updates to shared state are mediated by atomic primitives. We rely on *compare-and-swap* (`CAS`): `CAS`$(addr, old, new)$ atomically writes new to $addr$ if its current value is old, and returns whether it succeeded. `CAS` is the building block for the lock-free union-find we use in subsection III-C. Contention, where two threads attempt to update the same location simultaneously, manifests as a failed `CAS` that is retried against the new value, never as blocking.

Finally, our algorithms are structured as a sequence of *bulk-synchronous* rounds [22]: each round performs a large parallel computation, all threads synchronize at a barrier, and the next round consumes the results. This structure makes per-round work and span easy to reason about and matches how new congruences are discovered in waves (subsection II-B). The key per-round primitive we use is `group_by`, also called *semisort* [23]: given an array of (key, value) pairs it returns the values grouped by key, in $O(n)$ work and $O(\log n)$ span on average. We use it to gather candidate-congruent terms in each round before unioning their classes.

III. ALGORITHMS

A. Sequential Baseline

Our baseline (Algorithm 1) is a simple yet efficient worklist algorithm based on one developed by Nieuwenhuis and Oliveras [24].

We maintain a *signature table* that maps every term to the representative of its equivalence class. It is implemented via a hashtable, where hashing a term entails hashing its function symbol and the equivalence-class identifiers of its immediate children. The signature table admits operations LOOKUP(e), which finds the representative of term e (if any), and ENTER(e), which enters term e into the signature table.

We also maintain a union-find data structure that maintains the congruence classes of every term. The algorithm begins by issuing a UNION for each input equality, so that the initial union-find reflects exactly those equalities.

Finally, we maintain the *parents* of every equivalence class: the set of terms that are parents of at least one member of the equivalence class.

The algorithm then seeds a FIFO *worklist* of terms whose congruence classes may be coarser than those maintained by the union-find. It is seeded with all non-leaf terms (i.e. function applications) in reverse-topological order. Reverse-topological ordering ensures that the algorithm terminates in a single pass when the quotient DAG is acyclic, and serves as a good heuristic otherwise.

The algorithm then iterates until the worklist is empty. A term e is popped from the front of the worklist, and then inserted into the signature table. If there is a collision—meaning that a representative term e' with that signature already exists in the table—then the congruence classes of e and e' are merged. Then the parents of the *losing* term—that which is not the representative of the merged class—are added to the worklist if they are not already present. This is implemented in Algorithm 1.

Algorithm 1 Sequential worklist congruence closure algorithm

```
1: function CONGRUENT(X, Y)
2:     if X = f(M_1, ..., M_n) and Y = f(N_1, ..., N_n) then
3:         return FIND(M_i) = FIND(N_i) for all i = 1, ..., n
4:     else
5:         return false
6: function SEQUENTIALCLOSE(Eq)
7:     ▷ Eq = {e_1 = e'_1, ..., e_w = e'_w}
8:     for (u, v) ∈ Eq do
9:         UNION(u, v)    ▷ Add initial unions to union-find
10:    class_preds ← array indexed by equivalence-class id
11:    for every term e do
12:        for every child c of e do
13:            Append e to class_preds[FIND(c)]
14:    Let sig_table be a signature table
15:    Let worklist be a FIFO worklist of nodes
16:    Insert all non-leaf terms into worklist in reverse-
       topological order
17:    while Worklist is not empty do
18:        Pop node e from Worklist
19:        if sig_table contains e then
20:            rep ← sig_table.LOOKUP(e)
21:            ▷ New representative of class
22:            winner ← UNION(rep, e)
23:            loser ← rep if winner = e else e
24:            for parent ∈ class_preds[loser] do
25:                ▷ Merge parents of loser into winner
26:                Add parent to class_preds[winner]
27:                Add parent to Worklist
```

We tested a number of other sequential algorithms and found this one to be the most efficient. The other sequential algorithms we tested are included below.

1) Topological Sort: One approach is to topologically sort the terms, i.e. sort them in an order where each child must come before its parents. Then you can add each term to a signature table one at a time. If a term is already equal to a term in the signature table, you can union their equivalence classes in the union-find data structure. If there were any unions at all, we must iterate through all the terms again. We must iterate this until a fixed point is reached, since merging two equivalence classes can cause new merges to be possible. This approach is sometimes efficient, but breaks down when many iterations must be done.

2) DST: We also implemented a practical version of the Downey-Sethi-Tarjan (DST) algorithm [25] that achieves $O(n \log n)$ work. We made some changes, including eliminating the outdegree-2 transformation that curries each term into a binary term by representing an n-ary term as a tree of binary terms. Additionally, we represented the signature table as a hashtable instead of a trie, which is more performant in practice (although it forfeits the deterministic work bound).

B. Bulk-Synchronous Parallel Algorithm

The remainder of this section describes our bulk-synchronous-parallel approach to computing congruence closure. We build on two workhorses, each covered in its own subsection below: a concurrent union-find data structure (subsection III-C) used to maintain congruence classes, and *semisort* (subsection III-D), a parallel primitive offered by PARLAYLIB [16] that we use to group congruent terms. subsection III-E then puts these components together into our rounds-based algorithm.

C. Concurrent Union-Find

We employ a simple concurrent union-find presented by Alistarh et al. [15]. In their experiments, they study various different union-find algorithms, and find that most variants perform comparably across their evaluation. In particular, more theoretically efficient variants (like those employing randomized linking [26]) do not achieve better practical performance. We therefore chose a simple variant with confidence that it would perform well in practice, even under contention.

The algorithm employs best-effort linking by rank. The structure is represented as an array *UF*, where $UF[i]$ is the index of the parent of node i if i is not a root, or its rank otherwise. The high-order bit distinguishes whether the entry is another node or the rank of a root. We now describe the two operations it supports: FIND and UNION:

Find. A FIND(u) returns a pair $(root, rank)$ comprising the representative of the equivalence class of u and its rank. It first checks whether $UF[u]$ is a rank; if so, it returns a pair comprising u and that rank. Otherwise, $UF[u]$ is its parent, and it invokes FIND on the parent. It then attempts to path compress via a CAS, which may fail if the path has already been compressed. In pseudocode we write FIND(u) for the root alone; the rank is used only internally by UNION.

Union. A UNION(u, v) operation unions the equivalence classes of u and v. It first computes the roots and ranks of u and v via FIND. It then checks whether the representatives are equal; if so, u and v are already part of the same equivalence class and it returns immediately. Otherwise, it compares the ranks of u and v. If one rank is strictly greater than the other, then the algorithm attempts to set the parent of the lower-rank node to the greater-rank node with a CAS. This may fail if the parent of the lower-rank node has been updated in the meantime, in which case the UNION restarts, recomputing the roots and ranks with a fresh FIND before retrying.

If both ranks are equal, then it selects one node arbitrarily to be the parent and attempts to update the child's parent pointer via a CAS, again retrying if it fails. It then attempts to increment the rank of the node chosen to be the parent, which again may fail if another UNION operation interferes. If it fails, it means that the node chosen to be a parent is no longer a root, and thus has no rank. Thus incrementing the rank can fail, and linking by rank is best-effort. In every case—including when the two classes were already equal—UNION returns the representative of the merged class.

D. Parallel Semisort

At each round, we want to group congruent terms based on newly discovered equalities. We use the *semisort* of Gu, Shun, Sun, and Blelloch [23]: given an array of (hash, term) pairs, semisort permutes the array so that all pairs sharing the same hash are adjacent, without imposing a total order across distinct hashes. This relaxation lets semisort run in $O(n)$ work and $O(\log n)$ span on average, strictly better than a comparison sort. We use it to gather congruence classes at each round. In our pseudocode we write GROUPBY$(\sim, A)$ for the operation that partitions A into groups under an equivalence relation $\sim$ on A.

E. Rounds-Based Congruence Closure

At each round, we maintain a set $Work$ of the representatives dethroned by the previous round's merges (initially, the terms appearing in base equalities). A round begins by folding the parent lists of these dethroned roots into their new representatives, maintaining a mapping $Parents$ from each class to the parents of its members: if the congruence class of a term e has changed, then the congruence class of $f(e)$ may change as well. The *frontier* of the round—the parents of the classes in $Work$—is then grouped into congruent terms via semisort (the `group_by` operation in PARLAYLIB). Each group *spanning* more than one class (i.e., whose members do not all share one representative) is merged in the union-find, and its classes' dethroned pre-merge roots form the next round's $Work$. These rounds continue until there is no longer any work to do.

We call this algorithm PARENTCC, since it maintains an explicit parent list (the $Parents$ mapping) for each congruence class. Algorithm 2 gives its pseudocode, decomposed into two functions. Both functions rely on the predicate CONGRUENT, which checks whether two terms are congruent based on the equivalence classes in the union-find: two terms are congruent if they have the same function symbol and, under the current union-find, each of their children has the same representative (as defined in Algorithm 1).

The MERGECONGRUENCECLASS function is used to merge the equivalence classes of different terms. Specifically, given a sequence S of terms, it will union the classes of every term in S. We take a simple divide-and-conquer approach; we merge the left and right halves (in parallel), and then UNION the representatives of the two classes.

a) Correctness: We sketch why PARENTCC is correct: (1) it terminates, and (2) on termination the union-find partitions $\mathcal{T}$ into exactly the congruence classes of $\equiv$. The union-find is linearizable [15], so a round's concurrent UNIONs behave as unions performed in some sequential order—and since the partition generated by a set of unions is independent of that order, each round produces a deterministic partition. The barrier between grouping and merging ensures every signature is computed by FIND against a union-find whose partition no thread is concurrently modifying.

Termination. Beyond the initial seed, a term enters $Work$ only when a merge dethrones it as a representative. A term is dethroned at most once across the run, so at most $|\mathcal{T}|-1$ terms ever enter $Work$; a round in which every group lies within a single class adds nothing to $Work$, and the loop halts.

Soundness. We maintain the invariant that every union-find class is contained in a congruence class of $\equiv$. It holds after the initial UNIONs, which merge only base equalities. Assume it holds when a round computes its signatures, and suppose $e = f(M_1, \ldots, M_n)$ and $e' = g(N_1, \ldots, N_m)$ are grouped together. Their signatures agree, so $f = g$, $n = m$, and M_i and N_i shared a class when signatures were computed; by the invariant $M_i \equiv N_i$, hence $e \equiv e'$ by congruence. The round's unions glue classes only along such grouped pairs, so by transitivity of $\equiv$ every resulting class remains within a class of $\equiv$.

Completeness. Conversely, every pair related by $\equiv$ shares a class on termination. Let $\sim$ be the relation "shares a union-find class on termination." Then $\sim$ is an equivalence relation,

Algorithm 2 PARENTCC: parents-based bulk-synchronous algorithm

```
1:  function MergeCongruenceClass(Class)
2:      if Class = {e} then
3:          return e   ▷ Singleton class, return representative
4:      n ← |Class|
5:      L ← Class[0, ..., n/2]
6:      R ← Class[n/2, ..., n]
7:      ▷ Merge left and right halves, get their representatives
8:      l, r ← MergeCongruenceClass(L) ‖ MergeCongruenceClass(R)
9:      return Union(l, r) ▷ Union left and right reps, return root
10: function ParentCC(Eq)
11:     ▷ Eq = {e_1 = e'_1, ..., e_w = e'_w}
12:     ▷ Parents[c] initially holds the immediate parents of term c
13:     parfor (u = v) ∈ Eq do
14:         Union(u, v)
15:     ▷ Initialize Work with terms appearing in equalities
16:     Work ← {e_1, e'_1, ..., e_w, e'_w}
17:     while Work is not empty do
18:         ▷ Fold parent lists of merged classes into their new root
19:         parfor c ∈ Work with c ≠ Find(c) do ▷ c is not the representative of its class
20:             Parents[Find(c)] ← Parents[Find(c)] ∪ Parents[c]
21:         Frontier ← ⋃_{c∈Work} Parents[Find(c)]
22:         groups ← GroupBy(Congruent, Frontier)
23:         Work ← ∅
24:         parfor g ∈ groups spanning more than one class do
25:             ▷ Merge every term within congruence class
26:             rep ← MergeCongruenceClass(g)
27:             Work ← Work ∪ {pre-merge roots of g's classes} \ {rep}
```

and the initial UNIONs ensure $E \subseteq \sim$, so it suffices to show that $\sim$ is closed under congruence: $\equiv$, the *smallest* such relation (Definition 2), is then contained in $\sim$. Suppose $e = f(M_1, \ldots, M_n)$ and $e' = f(N_1, \ldots, N_n)$ with $M_i \sim N_i$ for every i. Each pair M_i, N_i is merged at some round; consider the round performing the last such merge (counting the initial UNIONs as round zero, which seed $Work$ with the terms of Eq). That merge places the losing representatives in $Work$, so the next round folds their parent lists into the new root—maintaining the invariant that a class's root holds the parents of all its members—and its frontier contains the merged class's parents, including e and e'. There they receive identical signatures and are merged (or already share a class). Classes never shrink, so $e \sim e'$.

F. Filtering-Based Algorithm (FILTERCC)

The previous algorithm, like the sequential baseline, explicitly maintains the parents associated with each congruence class, which is expensive due to frequent allocations. We now present an algorithm that achieves the same effect without keeping these parent lists.

Instead of explicitly maintaining lists, we instead maintain which classes are "dirty" at a given round—i.e., were involved in a merge during the prior round. We then filter all terms to retain those with a child in a dirty class, and propagate these terms to the next round to be semisorted and recanonicalized. This avoids the cost of maintaining parent lists, but also requires scanning all nodes every round. We have found that this tradeoff is generally worthwhile on most workloads. Algorithm 3 gives the pseudocode.

FILTERCC is correct by the same argument as PARENTCC: it computes the same frontier—every term with a child in a class merged since the last round—by filtering all terms for dirty children rather than consulting parent lists.

Algorithm 3 FILTERCC: filtering-based bulk-synchronous algorithm

```
1:  function FilterCC(Eq)
2:      ▷ Eq = {e_1 = e'_1, ..., e_w = e'_w}
3:      parfor (u = v) ∈ Eq do
4:          Union(u, v)
5:      ▷ Dirty flags, indexed by term id (one per term)
6:      dirty[Find(u)] ← true for each u appearing in Eq
7:      Frontier ← {g(... u ...) | dirty[Find(u)]}
8:      while Frontier is not empty do
9:          dirty ← [false | 0 ≤ i < |T|]
10:         groups ← GroupBy(Congruent, Frontier)
11:         parfor g ∈ groups spanning more than one class
            do
12:             ▷ Merge every term within congruence class
13:             rep ← MergeCongruenceClass(g)
14:             dirty[rep] ← true
15:         Frontier ← {g(... u ...) | dirty[Find(u)]}
```

G. Implementation Details

We have found that several implementation details significantly improve the performance of both algorithms. Although we use PARLAYLIB's scheduler and parallel primitives throughout, we do not use its semisort implementation directly. Instead, we use PARLAYLIB's integer sort to group the nodes by hash and then *sequentially* bucket the resulting array manually—this requires detecting hash collisions. This is more performant for small buckets. PARLAYLIB's semisort buckets in parallel, which is advantageous for large buckets, but not the small ones we typically encounter in our workloads.

Several implementation choices also had significant impact on the performance of the sequential algorithm. Firstly, we maintain different signature tables for terms of arities 1, 2, 3, and 4. Terms of greater arities fall into the general signature table. Heterogeneous signature tables require storing terms of different arities (and hence size), and require dynamic allocations for each canonical signature placed into the table. Thus for the common case (terms of small arity), it is advantageous to avoid this overhead. Even for the heterogeneous table, we use a bump allocator to reduce the number of allocations.

IV. EVALUATION

In this section, we evaluate our parallel congruence closure algorithms on a number of benchmarks in order to answer the following research questions:

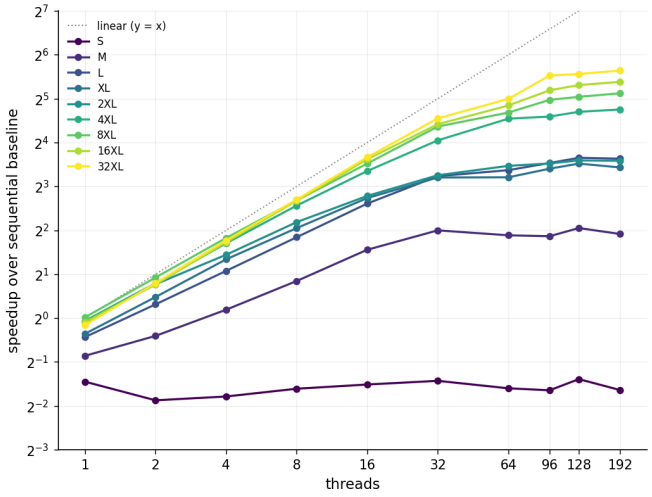
- **RQ1:** Do our two parallel congruence closure algorithms achieve a meaningful speedup over a sequential implementation and how does this speedup scale with the number of threads?
- **RQ2:** What characterizes the type of benchmark that our parallel congruence closure algorithms perform well on?
- **RQ3:** How do our parallel congruence closure algorithms perform on real-world benchmarks?

We demonstrate the efficacy and limitations of our approach through a wide selection of benchmarks on various workloads. In subsection IV-A, we discuss a randomly generated benchmark suite. In subsection IV-B, we discuss a synthetic benchmark suite that allows us to dial up the depth and width of the benchmark to better understand how these parameters affect the performance of our algorithms. Finally, in subsection IV-C, we discuss a set of circuit equivalence benchmarks [4].

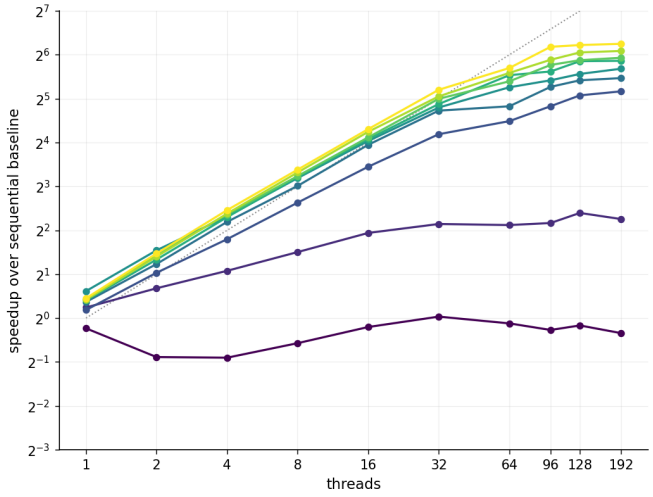
All experiments were run on a 96-core Amazon Web Services `c8i-metal` instance with Intel(R) Xeon(R) 6975P-C (3.9 GHz) and 384 GB memory. Each core is 2-way hyperthreaded, giving 192 hyperthreads. The machine runs Ubuntu 26.04 LTS, and the code was compiled using g++ 15.2.0 with -O3.

For each benchmark, we perform 1 warmup run and 5 timed runs, and we report the geometric mean of the runtime across the 5 timed runs. Throughout this section we write T for the number of software threads used; in particular, T=1 denotes the single-threaded configuration of a parallel algorithm.

A. Random Benchmarks



(a) Speedup of ParentCC compared to sequential baseline.



(b) Speedup of FilterCC compared to sequential baseline.

Fig. 2: Random benchmarks speedup using ParentCC and FilterCC (log-log scale).

workload	n_{leaves}	n_{compound}	n_{merges}	depth	n_{fns}
`small`	1 000	10 000	2 000	3	4
`medium`	10 000	200 000	20 000	3	8
`large`	50 000	1 000 000	100 000	3	16
`xl`	100 000	2 000 000	200 000	3	16
`2xl`	200 000	4 000 000	400 000	3	16
`4xl`	400 000	8 000 000	800 000	3	16
`8xl`	800 000	16 000 000	1 600 000	3	16
`16xl`	1 600 000	32 000 000	3 200 000	3	16
`32xl`	3 200 000	64 000 000	6 400 000	3	16

TABLE I: Random workload parameters. We have n_{fns} function symbols, each of arity 2. Each workload has n_{leaves} `x`-leaves plus n_{leaves} `y`-leaves at the base, n_{compound} compound terms of depth d built from uniformly-chosen function symbols, and n_{merges} random initial equalities, each pairing one `x`-leaf with one `y`-leaf.

Our first set of benchmarks are randomly generated based on several parameters. We first define a set of function symbols F, which has n_{fns} total functions. Then we have a set of terms $\mathcal{T}$ that contains n_{leaves} `x`-leaves and n_{leaves} `y`-leaves, plus n_{compound} compound terms of *depth* d (i.e. with d nested function applications). We initially start with the equalities $E = \{x_{i_1} = y_{i_1}, \ldots, x_{i_{n_{\text{merges}}}} = y_{i_{n_{\text{merges}}}}\}$, i.e. n_{merges} random equalities where each pair contains one `x`-leaf and one `y`-leaf. Essentially, this setup divides the set of leaves into two groups and merges a random subset of them across the groups. We stratify our workloads into different size classes, which are shown in Table I.

Figure 2 shows the speedup of our parallel algorithms over the sequential baseline. We observe that on workloads with more nodes, our algorithms achieve close to linear speedup, plateauing between 32 and 64 threads. For instance, on `32xl`, FilterCC (resp. ParentCC) achieves a speedup over sequential of 36.8× (23.4×) on 32 threads, 51.9× (31.9×) on 64 threads, and 76.1× (50.0×) on 192 threads.

While both ParentCC and FilterCC achieve similar parallelization on larger workloads, FilterCC's single-thread runtime is much better than ParentCC's. This is because ParentCC must maintain a parent list for each term, which is expensive due to the frequent allocations required. The sequential algorithm also maintains parent lists, but is able to avoid duplicates.

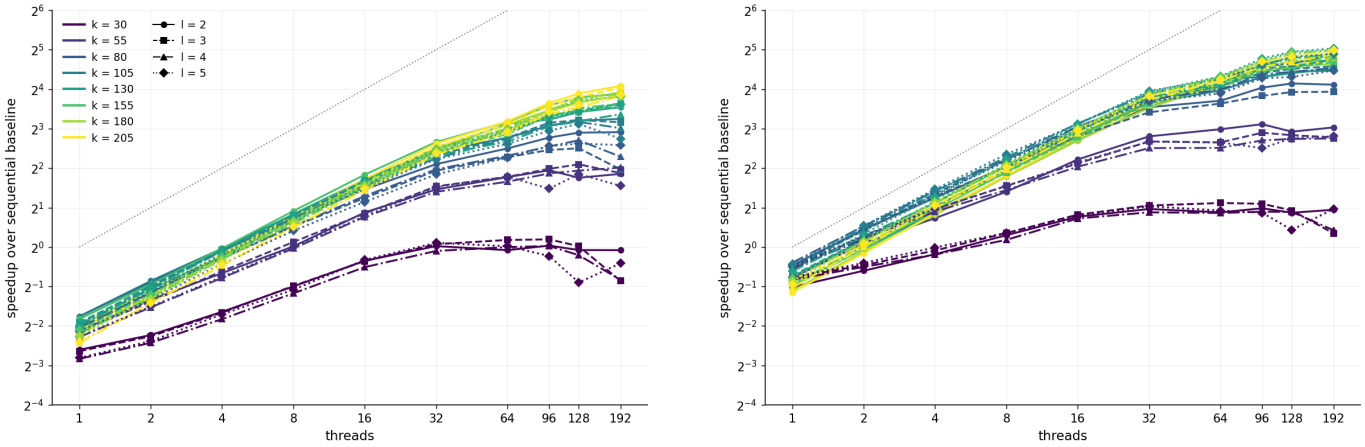
Note that on sufficiently large workloads we observe *superlinear* speedups, e.g., FilterCC on `32xl` reaches 36.8× on 32 threads. We believe this is because FilterCC has fewer data structures to maintain, so on larger workloads the sequential version suffers more from cache misses and other memory overhead. In particular, maintaining parent lists in the sequential algorithm (and in ParentCC) has poor spatial locality, since the children of a term may be scattered far from the parent in memory.

B. Synthetic Benchmarks

To better understand the relationship between the depth (i.e. number of rounds of congruence closure) and the width (i.e. number of merges per round) of the benchmark and the performance of our algorithms, we design a synthetic benchmark suite that allows us to dial up these parameters.

Our benchmark suite is parameterized by a width parameter k and a g-tree fan-in l, and is constructed in three layers:

- **Leaves.** $2k$ leaf constants $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$.
- **f-layer.** For every triple $(i, j, m) \in \{1, \ldots, k\}^3$ we add the terms $f(a_i, a_j, a_m)$ and $f(b_i, b_j, b_m)$, contributing $2k^3$ terms in total.
- **g-tree.** On each of the a- and b-sides we build a balanced l-ary tree of g-applications whose leaves are the k^3 f-terms on that side. Each internal g-node has exactly l children, and the tree has height $\lceil \log_l(k^3) \rceil$. The two sides share no g-nodes.



(a) Speedup of ParentCC compared to sequential baseline.

(b) Speedup of FilterCC compared to sequential baseline.

Fig. 3: Synthetic benchmarks speedup using ParentCC and FilterCC.

The initial unions are $a_i = b_i$ for $i \in \{1, \ldots, k\}$. After congruence closure has propagated these unions through the f-layer there are k^3 congruences of the form $f(a_i, a_j, a_m) \equiv f(b_i, b_j, b_m)$, which must in turn propagate up the two g-trees until the two roots become equal.

The width parameter k controls how many merges occur at the first congruence-closure round above the leaves: a larger k means a much larger frontier (k^3 merges) of mutually-independent f-equalities, all of which can be discharged in parallel. The fan-in l controls the height of the g-tree above the f-layer: each level reduces the active frontier by a factor of l, so a larger l produces a shallower closure (fewer rounds) at the cost of wider fan-in per g-node. We sweep $k \in \{30, 55, 80, 105, 130, 155, 180, 205\}$ and $l \in \{2, 3, 4, 5\}$.

Figure 3 reports the speedup of both algorithms over the sequential baseline across this sweep. We see that the width of the benchmark has a much larger impact on the speedup than the depth. Holding k and thread count T fixed, the parallel speedup varies by only $\sim$20% across the four tree depths we measured (median max/min over l of $1.21\times$ for FilterCC and $1.20\times$ for ParentCC), demonstrating that depth has negligible impact on parallel performance within this benchmark family. This ratio is roughly constant across thread counts, ranging from $1.15\times$ at $T{=}8$ to $1.31\times$ at $T{=}128$ for FilterCC, and from $1.13\times$ at $T{=}4$ to $1.29\times$ at $T{=}192$ for ParentCC. Holding l and thread count fixed, on the other hand, varying k produces a much wider speedup spread (median max/min over k of $5.97\times$ for FilterCC and $4.87\times$ for ParentCC). Unlike the depth ratio, the width ratio grows sharply with thread count: at $T{=}1$ it is $1.49\times$ for FilterCC and $1.78\times$ for ParentCC, but by $T{=}192$ it reaches $20.2\times$ and $23.2\times$ respectively.

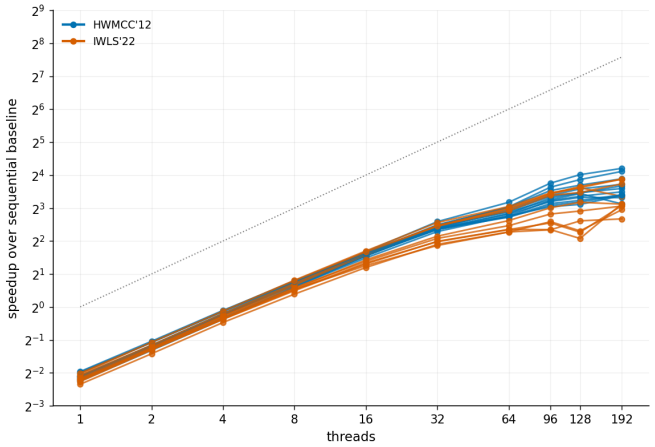
Similar to the random workloads, our algorithms achieve near-linear speedup over the sequential version up to a plateau between 32 and 64 threads. However, the single-core baseline is noticeably worse than the sequential algorithm: at $k{=}205$, $l{=}2$, FilterCC at $T{=}1$ is $2.22\times$ slower than the sequential algorithm and ParentCC at $T{=}1$ is $4.25\times$ slower. On the same workload, FilterCC (resp. ParentCC) achieves a speedup over sequential of $6.79\times$ ($3.30\times$) on 16 threads, $12.3\times$ ($6.16\times$) on 32 threads, $17.0\times$ ($9.09\times$) on 64 threads, and $26.4\times$ ($17.0\times$) on 192 threads.

C. Circuit Equivalence Benchmarks

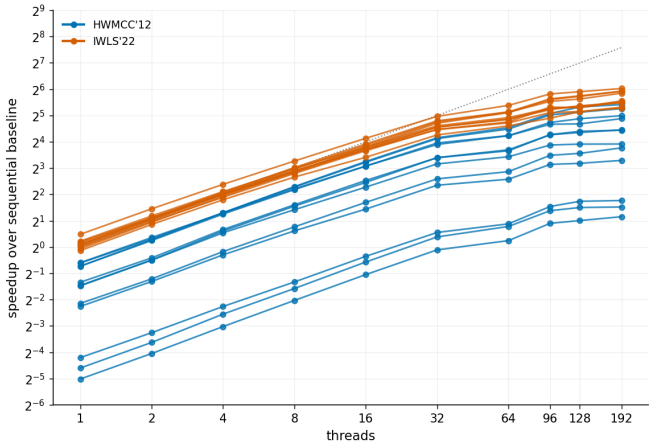
These benchmarks come from the clausal congruence closure work of Biere et al. [4], which (as discussed in subsection II-B) uses congruence closure over recovered gates to simplify miters before SAT solving. They implemented their algorithm as a preprocessing pass in Kissat [5] and evaluated on two benchmark sets: 10 hand-curated miters from the IWLS'22 paper [27], considered hard for monolithic CNF-level SAT sweeping, and a derivative of the 2012 Hardware Model Checking Competition [28] that compares each And-Inverter Graph against an optimized variant.

For our evaluation we use the same circuit sources, but rather than running congruence closure as a preprocessing pass within Kissat we extract its intermediate gate set and run our parallel congruence closure implementations on it directly. Concretely, we instrument Kissat's gate extractor to dump every candidate gate (AND/XOR/ITE) and run our congruence closure implementations on these gates. We use all 10 IWLS'22 miters, and from the HWMCC'12 set we pick the twelve hardest instances by sequential congruence closure time, giving 22 files in total.

Figure 4 reports the speedup of both algorithms over the sequential baseline on these benchmarks as the thread count T is varied. On the circuit-equivalence benchmarks, FilterCC and ParentCC achieve near-linear speedup up to a plateau between 32 and 64 threads on most workloads. However, unlike on the random and synthetic benchmarks, the single-thread runtime of FilterCC varies dramatically across files. On the 22 files we measured, its runtime at $T{=}1$ ranges from $0.71\times$ the sequential runtime on `h5m-n04-ands.gates`—the file with the largest single-thread advantage over sequential—up to



(a) Speedup of ParentCC compared to sequential baseline.



(b) Speedup of FilterCC compared to sequential baseline.

Fig. 4: Speedup of ParentCC and FilterCC on circuit equivalence benchmarks.

32.3× the sequential runtime on `6s104-xits-opt.gates`. ParentCC, by contrast, is consistently $\sim 4\times$ slower than sequential at $T{=}1$ regardless of file.

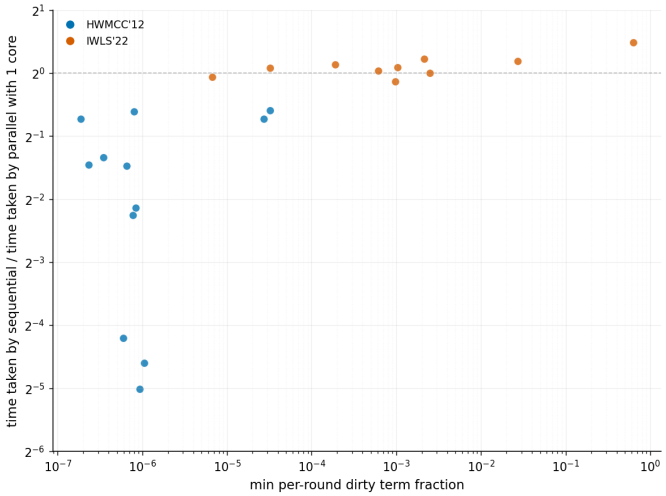


Fig. 5: Ratio of sequential baseline to FilterCC runtime at $T{=}1$ vs. the smallest fraction of dirty terms across rounds.

Figure 5 explains this spread by plotting the speedup of FilterCC against the *minimum per-round dirty term fraction*: for each round r of the algorithm we record $\mathrm{dirty}[r]/\mathrm{live}[r]$, where $\mathrm{dirty}[r]$ is the number of terms the round marks as needing re-canonicalization and $\mathrm{live}[r]$ is the number of equivalence classes still alive after the preceding round's unions; we then take the minimum over all rounds the closure executes. A small minimum is bad for FilterCC because every round, regardless of how few terms are actually dirty, pays the full $O(\mathrm{live})$ cost of scanning all live terms to filter out the dirty ones — so a workload with even one near-empty round wastes work proportional to all live terms for very few useful unions.

V. Discussion

A. Answer to Research Questions

We answer the research questions as follows:

- **RQ1:** Our parallel congruence closure algorithms achieve meaningful speedups on random benchmarks, a synthetic benchmark suite designed to stress-test our algorithms, and a set of circuit equivalence benchmarks. We scale well up to 32 cores (FilterCC on `32x1` achieves 27× self-speedup at $T{=}32$ relative to its $T{=}1$ baseline, 36.8× over the sequential baseline), but begin to plateau beyond that, especially on smaller benchmarks.
- **RQ2:** Our algorithms tend to perform better on benchmarks with a large number of merges per round. Within our synthetic benchmark family, width has a much larger effect on parallel speedup than depth.
- **RQ3:** Our algorithms provide a meaningful speedup over a sequential implementation on the circuit equivalence benchmarks. It is difficult to evaluate our algorithms on SMT or equality saturation benchmarks, as congruence closure is part of a larger tool and thus difficult to isolate. We discuss integration prospects in subsection V-B.

B. Future Work

In the future, we would like to integrate our algorithms into a state-of-the-art SMT solver or equality saturation engine. This requires significant work. For instance, implementing our algorithm in an SMT solver requires: (1) adapting the batch algorithm to an incremental setting with backtracking, and (2) managing interaction with other theory solvers that share the e-graph. Even on pure equality with uninterpreted functions (EUF) benchmarks, an end-to-end SMT runtime comparison would conflate congruence closure with SAT decisions, propagation, and conflict analysis.

It is not clear what types of benchmarks would benefit from parallel congruence closure in an SMT solver or equality

saturation engine. For instance, we would expect that these benchmarks would have to have large enough congruence closure problems to benefit from parallelization, and that the congruence closure problems would have to be sufficiently wide. A practical dispatcher could choose between the sequential and parallel implementations based on available cores, problem size, and estimated congruence width.

VI. Conclusion

We presented ParentCC and FilterCC, two parallel congruence closure algorithms based on a bulk-synchronous design over a lock-free concurrent union-find. Despite congruence closure being P-complete, both algorithms achieve near-linear speedup up to 32 cores on random, synthetic, and circuit-equivalence benchmarks. We characterized the workload properties that govern parallel scaling: within our synthetic benchmark family, congruence *width* has a much larger impact on parallel scaling than *depth*.

VII. AI Statement

We used Claude Opus 4.7 in Agent Mode using Claude Code to aid the development of our code. We also used generative AI to assist with writing, including standard grammar, formatting, modifying tikz figures, editing, revisions, fleshing out proof details, and making passes over the paper.

Data availability

The artifact(s) associated with this paper is/are available at Zenodo under the following DOI: 10.5281/zenodo.20133810.

References

[1] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 185, pp. 825–885.

[2] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Principles of Programming Languages (POPL)*, 2009.

[3] P. C. Kanellakis and P. Z. Revesz, "On the relationship of congruence closure and unification," *Journal of Symbolic Computation*, vol. 7, no. 3–4, pp. 427–444, 1989.

[4] A. Biere, K. Fazekas, M. Fleury, and N. Froleyks, "Clausal Congruence Closure," in *27th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2024.

[5] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, 2020.

[6] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," in *Principles of Programming Languages (POPL)*, 2021.

[7] Y. Zhang, Y. R. Wang, O. Flatt, D. Cao, P. Zucker, E. Rosenthal, Z. Tatlock, and M. Willsey, "Better together: Unifying Datalog and equality saturation," *Proc. ACM Program. Lang. (PLDI)*, 2023.

[8] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[9] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2022.

[10] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[11] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying Rust programs using linear ghost types," *Proc. ACM Program. Lang.*, no. OOPSLA, pp. 286–315, 2023.

[12] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "CoSA: Integrated verification for agile hardware design," in *Formal Methods in Computer Aided Design (FMCAD)*, 2018.

[13] Lean Prover Community, "The grind tactic," https://lean-lang.org/doc/reference/latest/The--grind--tactic/, 2025, Lean 4 Reference Manual; accessed 2026-05-01.

[14] G. Nelson and D. C. Oppen, "Fast decision procedures based on congruence closure," *Journal of the ACM*, vol. 27, no. 2, pp. 356–364, 1980.

[15] D. Alistarh, A. Fedorov, and N. Koval, "In search of the fastest concurrent union-find algorithm," in *23rd International Conference on Principles of Distributed Systems, OPODIS*, 2019.

[16] G. E. Blelloch, D. Anderson, and L. Dhulipala, "ParlayLib – a toolkit for parallel algorithms on shared-memory multicore machines," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[17] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1980.

[18] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, 1975.

[19] G. E. Blelloch, L. Dhulipala, and Y. Sun, "Introduction to parallel algorithms (draft)," https://www.cs.cmu.edu/~guyb/paralg/paralg/parallel.pdf, 2026, lecture notes, Carnegie Mellon University.

[20] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM*, vol. 21, no. 2, pp. 201–206, 1974.

[21] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[22] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[23] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch, "A top-down parallel semisort," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2015.

[24] R. Nieuwenhuis and A. Oliveras, "Proof-producing congruence closure," in *Term Rewriting and Applications (RTA)*, ser. Lecture Notes in Computer Science, J. Giesl, Ed., vol. 3467. Springer, 2005, pp. 453–468.

[25] P. J. Downey, R. Sethi, and R. E. Tarjan, "Variations on the common subexpression problem," *Journal of the ACM*, vol. 27, no. 4, pp. 758–771, 1980.

[26] S. V. Jayanti and R. E. Tarjan, "A randomized concurrent algorithm for disjoint set union," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 2016.

[27] H.-T. Zhang, J.-H. R. Jiang, A. Mishchenko, and L. G. Amarù, "Improved large-scale SAT sweeping," in *Proc. 31st International Workshop on Logic & Synthesis (IWLS)*, 2022.

[28] A. Biere, M. Heule, M. Järvisalo, and N. Manthey, "Equivalence checking of HWMCC 2012 circuits," in *Proc. of SAT Competition 2013 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, A. Balint, A. Belov, M. Heule, and M. Järvisalo, Eds., vol. B-2013-1. University of Helsinki, 2013, p. 104.