Viability of Parallelization of Saturation and Extraction Algorithms for E-Graphs

Szymon Zyguła* and Krzysztof Kaczmarski

Warsaw University of Technology, Faculty of Mathematics and Information Science
szymon.zygula@pw.edu.pl
krzysztof.kaczmarski@pw.edu.pl

Abstract. Equivalence graphs (e-graphs) are data structures used in rewriting systems for representing expressions from a language, e.g. a programming language. An algorithm called *equality saturation* expands an e-graph representing a single expression to an e-graph representing all expressions equivalent to the starting one in accordance with a set of rewriting rules. Although effective, this process is computationally expensive, and parallelization remains largely unexplored. We evaluate parallel strategies for saturation and extraction, demonstrating significant speedups in matching and extraction phases, while highlighting fundamental limitations in parallelizing rule application within current architectures. Additionally, we introduce a general-purpose parallel Union-Find data structure featuring efficient lock-free path compression.

Keywords: E-Graphs, Equality Saturation, Rewriting, Parallel Union Find

1 Introduction and Related Work

Equivalence graphs (e-graphs) are data structures that represent multiple equivalent expressions, making them useful for program optimization, algebraic simplification, computer algebra [6], and numerical accuracy optimization [12].

E-graphs are used to represent expressions from some language, e.g. a programming language or the language of algebraic expressions. In contrast to traditional expression representation using trees, e-graphs can compactly represent multiple equivalent expressions. This makes e-graphs useful for optimization, where the goal is to find an equivalent expression minimizing a cost function. Such optimizations are done through a procedure called *term rewriting*: having a set of rewrite rules we apply them to the optimized expression until we achieve the desired result [2]. These rules can be, e.g. in the case of a programming language, loop unrolling or constant folding. A basic optimization algorithm applies a given rule set to the given expression until no rules can be applied anymore. The problem with this process is that it is *destructive*, i.e. application of one rule modifies the existing expression. This can lead to situations in which application of one rewrite rule prevents application of other rules.

* Corresponding author.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_49

Let us consider a classical example of rewriting in computer program optimization: rules replacing $x * 2$ with $x << 1$ and $(x * y)/y$ with x yield different results on $(x * 2)/2$ depending on order—$(x << 1)/2$ if shifting first, or optimal x if simplifying first. This *phase-ordering problem* [15] is often solved by heuristics or backtracking, but these are time- and memory-intensive. E-graphs solve this by representing multiple equivalent expressions non-destructively, applying rewrites until saturation, then extracting the optimal form via a cost function.

Despite their simplicity and better performance than backtracking, e-graphs are still vastly slower than traditional term rewriting systems with heuristics and rarely achieve saturation because explosive growth can make them too large to fit in memory [7]. Although slow saturation is a well-known problem, no known literature describes any attempts at tackling the problem by using multiple parallel threads. Parallelizing their operations introduces challenges such as managing dependencies in rule application, avoiding excessive synchronization overhead, and handling dynamic structural updates.

E-graphs, introduced by Nelson [10] for congruence closure in Automated Theorem Provers (ATP), can be seen as an extension of the Union-Find (UF) data structure [14, 3]. Tate et al. introduced *Equality saturation* [15] for program optimization to mitigate the phase-ordering problem. Nieuwenhuis and Oliveras presented efficient, human-readable *explanations* for expression provenance [11].

The popular *egg* library [16] is a flexible and extensible implementation of e-graphs and equality saturation. It introduced features like analysis data and deferred rebuilding, which improve performance by separating read and write operations; however, it is single-threaded by default, leaving parallelization unexplored. Our work improves egg in this dimension. De Moura and Bjørner developed an efficient e-matching algorithm for SMT solvers [5], which egg uses.

In our research we found out that in our parallel algorithms it is particularly useful to have a work-stealing scheduler, which is provided by the *Rayon* library [13] alongside *Crossbeam* [4].

Jayanti and Tarjan proposed concurrent Union-Find algorithms using atomic operations like Double Compare and Swap and Compare and Swap [8]. Alistarh et al. evaluated several concurrent UF algorithms [1], but they are not well suited for our needs. We designed a new lock-free parallel UF data structure. Although it was designed for the specific access patterns of e-graphs, it can be employed in other applications as well.

2 Parallel Saturation and Extraction Algorithms

We denote an e-graph by $G = (V, C)$, where V are the e-nodes, and C are the e-classes. Expressions and e-nodes are denoted by $S(c_1, c_2, \ldots, c_n)$, where S is the root symbol, and $c_1, c_2, \ldots, c_n$ are either subexpressions or e-classes. The cost of an expression $S(c_1, c_2, \ldots, c_n)$ is written as $k_S(k_1, k_2, \ldots, k_n)$, where k is a cost function and k_i is the cost of c_i.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_49

2.1 Parallel Saturation

The saturation algorithm's main loop [16] is split into 3 parts:

1. finding expressions to which each rule can be applied (FindMatchesPar),
2. applying the rewrite rules to the e-graph (ApplyMatchesPar),
3. rebuilding the e-graph.

Deferred rebuilding proves useful in parallelization, as it allows us to easily postpone some work requiring modification of the e-graph to when all threads have completed tasks which can be done in parallel without locking and data races. Given that the matching phase does not modify the e-graph structure, it is well-suited for parallelization. The parallel matching algorithm differs from the sequential algorithm in two respects: both the iteration over rewrite rules and, for each rule, the scan of candidate e-classes are performed in parallel, and the algorithm uses the parallel UF structure. When matching of expressions is performed, the UF is searched for e-classes of nodes, and so parallel path compression is performed.

The parallel application algorithm modifies the e-graph, which prevents effective parallelization of this part of saturation.

Our algorithm performs some work in parallel and defers other tasks to a later stage, where they are executed sequentially. The rule application algorithm begins by spawning an *applier thread* and a *manager thread*. The applier thread (Fig. 1a) then spawns additional threads, each of which takes care of a different match from the matching step. They do so by creating e-nodes, which need to be added to the e-graph, and generating identifiers (IDs) for each of them. This is done using the parallel Union-Find data structure, described in detail in Section 2.3. When all nodes and IDs are created, they are sent to the applier thread through a multi-producer single-consumer (MPSC) channel, together with information on which e-classes unions need to be performed.

The manager thread (Fig. 1b) receives e-nodes which need to be added to the e-graph and IDs of e-classes to merge. When applier threads exit, the manager thread adds all nodes to the e-graph and performs necessary unions. This work consists almost exclusively of sequentially dependent writes to the e-graph, and thus cannot be easily parallelized.

Also, the parallel Union-Find developed in Section 2.3 is used for parallel set ID creation (Fig. 1a) and parallel path compression for node searching.

2.2 Parallel Extraction

The parallel cost calculation algorithm, which is the main part of extraction, is presented in Fig. 1c. The main difference in comparison to the sequential algorithm is the parallelization of the *for* loop with some synchronizations.

The variable b must be atomic, and assignments to it in parallel threads use relaxed memory ordering [9], denoted here by $\leftarrow_A$. Relaxed ordering provides the least amount of guarantees of all ordering types. In return, it works with the least amount of overhead and in some architectures its synchronization overhead

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_49

```
Input: G: e-graph, C: MPSC channel, M: list of rule matches
function ApplierThread(G, C, M)
    for all (E, r) = M_r ∈ M in parallel do
        ▷ IDs for each node in E' gener. here
        E' ← r(E)
        for all symbol S ∈ E' do
            ▷ Edges from n sent implic. as IDs
            Send (add, n) to C
        I_E ← ID of e-class of E
        I_E' ← ID of E'
        Send (union, (I_E, I_E')) to C
```

(a)

```
Input: G: e-graph, C: MPSC channel
function ManagerThread(G, C)
    N ← ∅ ▷ Buff. nodes to add to G
    U ← ∅ ▷ Buff. unions to perform on G
    ▷ Loop ends when all producers terminate
    for all message m incoming from C do
        if m = (union, (c1, c2)) then
            Add (c1, c2) to U
        else if m = (add, n) then
            Add n to N
    return N, U
```

(b)

```
Input: G: e-graph, k: cost function
function CalculateCosts(G = (V, C), k)
    ▷ Each element locked by a reader-writer
    K_C ← dictionary indexed by e-classes
    b ← true
    while b do
        b ← false
        for all c ∈ C in parallel do
            N_c ← ∅
            for all S(c1, c2, ..., cn) = v ∈ c do
                Acquire reader locks for K_C[c_i]_{i=1,...,n}
                if K_C[c1], ..., K_C[cn] are defined then
                    Insert (k_S(K_C[c_i]_{i=1,...,n}), v) into N_c
                Release reader locks for K_C[c_i]_{i=1,...,n}
            if N_c is not empty then
                ▷ Compared by first pair element
                (k0, v) ← smallest elem. of N_c
                Acquire reader lock for K_C[c]
                r ← K_C[c] is not defined or K_C[c] < k0
                Release reader lock for K_C[c]
                if r then
                    b ←_A true
                    Acquire writer lock for K_C[c]
                    K_C[c] ← (k0, v)
                    Release writer lock for K_C[c]
    return K_C
```

(c)

Fig. 1: Parallel rewrite rule application: applier thread (a), manager thread (b) and expression cost calculation in parallel extraction (c)

is zero [9]. Relaxed memory ordering is sufficient here because the variable b is not read when multiple threads operate on it, and implicit joining of the threads at the end of the parallel loop constitutes a memory barrier, making all threads see the same changes. Another necessary change is protecting each element of K_C with a separate *reader-writer* lock.

2.3 Parallel Lock-Free Union-Find

The UF in egg is a dynamic array where indices serve as node IDs. Parent pointers form trees for each e-class, where the root ID defines the e-class identifier.

Parallelizing the application phase is challenged by the Union-Find's dynamic growth. Sequentially, adding elements is a simple append, but capacity limits trigger reallocation. Concurrently, this is hazardous: if one thread triggers a memory move, others holding references face dangling pointers, memory corruption, or use-after-free errors. While a global lock ensures safety, it serializes access, negating parallelism benefits.

Our solution decouples ID reservation from array modification via a two-phase deferred update. First, PromiseSet uses a relaxed atomic fetch-and-add

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_49

on SET_COUNT [9] to reserve unique IDs lock-free. This is extremely efficient, typically requiring a single instruction. After parallel tasks finish, a single call to MAKEPROMISEDSETS performs an uncontended batch update, resizing the UF array once for all reserved IDs. This mechanism ensures safety by separating high-performance parallel reservation from sequential structure modification.

Path compression flattens UF trees during `find` operations, reducing amortized costs [3]. In classical settings, this creates data races. For example, if a thread traverses a path from node A to B to C while another concurrently changes B's parent to D, the first thread might read an inconsistent pointer. While locks prevent this, they serialize operations and eliminate parallelism benefits. Our solution uses atomic variables for each parent pointer. Atomic reads and writes ensure threads always see valid pointers within the UF structure. Crucially, relaxed memory ordering is sufficient: the fundamental invariant is that `find` must terminate at the canonical representative. The exact path taken or the visibility order of intermediate tree states does not affect correctness. This lock-free path compression avoids expensive synchronization like acquire-release semantics and is nearly overhead-free on architectures like x86 [9].

3 Results of the Parallel Method Evaluation

In the experiments we used: egg v.0.9.5 [16]; Rayon v.1.10.0 [13]; rustc v.1.79; Ubuntu Linux Kernel 5.4.0; AMD Rome 7742 CPUs with quota of 34 cores and 200GB of RAM allocated.

Evaluation methodology We evaluated 60 randomly generated expressions from egg's real-calculus and propositional-calculus test languages. Five expressions were generated for sizes from 10^2 to 10^6. Real calculus uses more symbols and floating-point constants, yielding less sharing and larger e-graphs; propositional calculus is more compact.

Saturation was benchmarked for the original single-threaded egg implementation and for our parallel algorithm with 1–32 physical cores. Rules relating to commutativity, associativity, and distributivity were omitted in the evaluation to avoid unbounded growth of e-graphs.

Execution time The resulting total running time as a function of the number of threads is shown in Fig. 2a, together with the running time of the sequential algorithm.

Scaling depends primarily on work granularity. Propositional calculus exhibits high subexpression sharing, so even large ASTs produce relatively small e-graphs; for small inputs, this leaves too little work to amortize parallel overhead, and even large inputs scale only modestly. Real calculus produces less sharing and therefore larger e-graphs, so large instances provide enough parallel work for strong scaling. For expressions of size 10^6, real calculus approaches inverse scaling with thread count, close to the best achievable under Amdahl's law.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_49

Fig. 2b and Fig. 2c show that matching benefits substantially from parallelism, whereas application does not outperform the sequential baseline. Fig. 2f and Fig. 2e explain this: most time is spent in the deferred, effectively single-threaded part dominated by e-graph updates. This suggests that substantially faster rule application would require a different e-graph architecture rather than a local optimization of the current one.

In our experiments, most of the saturation time lies in matching, which scales well and is practically worthwhile to parallelize. Instances where this is not true tend to exhibit explosive e-graph growth, making saturation unattractive regardless.

The extraction phase was evaluated on the same expressions as saturation. Fig. 2d shows that parallel extraction becomes beneficial mainly for large instances. The speedup is smaller than for matching but still substantial, and again depends on both input size and language.

4 Conclusions

This paper presents a systematic evaluation of intra-process parallelization for the core algorithms of equality saturation. Our findings reveal a sharp dichotomy in the potential for speedup: while read-intensive phases (matching and extraction) are highly amenable to parallelism, the write-intensive rule application phase is fundamentally constrained. The matching algorithm parallelizes the search over e-classes, achieving near-linear speedups for large, complex problems. The extraction algorithm, in turn, uses a combination of reader-writer locks and atomic operations to concurrently compute optimal expressions. Foundational to these successes is our development of a novel, lock-free parallel Union-Find data structure. This contribution provides two key techniques: efficient, concurrent path compression using relaxed-ordering atomics, and a deferred-update mechanism for concurrent set creation. These techniques yield a general-purpose primitive for high-performance disjoint-set operations with low overhead on common architectures like x86.

A key finding of our work concerns the rule application phase. Our analysis demonstrates that this phase, which is bottlenecked by the overall volume of sequentially dependent writes to the e-graph's core data structures, fails to achieve a significant speedup from parallelization. This suggests that achieving fully parallel saturation may be infeasible without a foundational redesign of the underlying data structure to better support concurrent writes.

Therefore, our findings suggest that future research in high-performance equality saturation should focus on searching for a new e-graph architecture with parallel processing model as a first-order design principle. This could involve exploring batch-based e-class merging, alternative approaches to maintaining analysis data, or fundamentally different expression representations that avoid the sequential bottlenecks we have identified. Our work highlights the value of this research direction.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_49

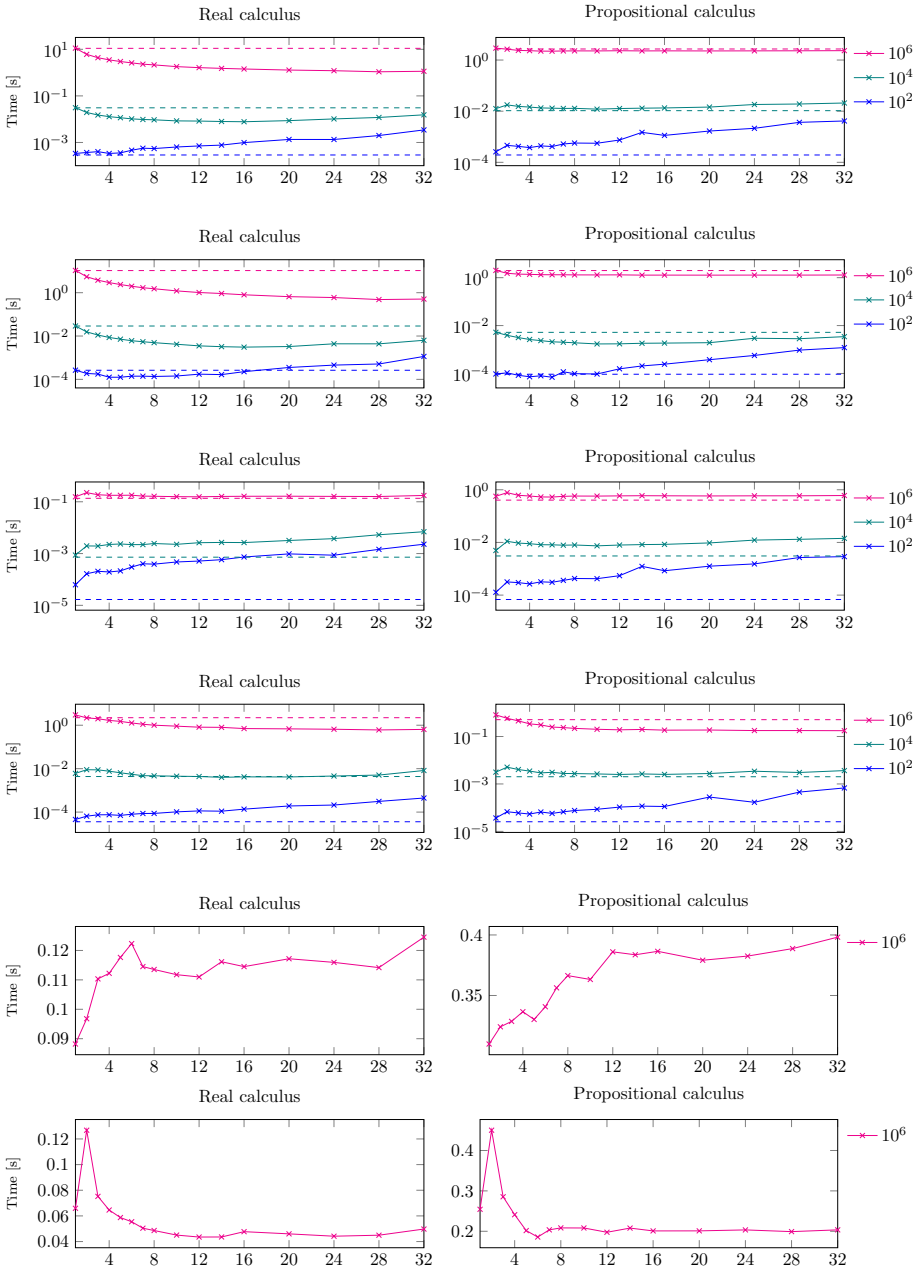

Fig. 2: Running times versus thread count: total saturation (a), matching (b), application (c), extraction (d), deferred work (e), and multithreaded work (f). Panels (a)–(d) are logarithmic; panels (e) and (f) are linear and show expressions of size 10^6. Solid lines show parallel runs by expression size; dashed lines show the sequential algorithm.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_49

References

1. Alistarh, D., Fedorov, A., Koval, N.: In Search of the Fastest Concurrent Union-Find Algorithm. In: 23rd International Conference on Principles of Distributed Systems (OPODIS 2019). Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2019)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Fourth Edition. MIT Press (2022)
4. Crichton, A., Kang, J., Turon, A., Endo, T., et al.: Crossbeam. https://crates.io/crates/crossbeam (2024), https://crates.io/crates/crossbeam, accessed 2024-06-07
5. De Moura, L., Bjørner, N.: Efficient E-Matching for SMT Solvers. In: Pfenning, F. (ed.) Automated Deduction – CADE-21, vol. 4603, pp. 183–198. Springer Berlin Heidelberg (2007)
6. Gowda, S., Ma, Y., Cheli, A., Gwóźdź, M., Shah, V.B., Edelman, A., Rackauckas, C.: High-performance symbolic-numerics via multiple dispatch. ACM Communications in Computer Algebra **55**(3), 92–96 (2022)
7. He, G., Singh, Z., Yoneki, E.: MCTS-GEB: Monte Carlo Tree Search is a Good E-graph Builder. In: Proceedings of the 3rd Workshop on Machine Learning and Systems. pp. 26–33. EuroMLSys '23, Association for Computing Machinery (2023)
8. Jayanti, S.V., Tarjan, R.E.: Concurrent disjoint set union. Distributed Computing **34**(6), 413–436 (2021)
9. Moiseenko, E., Podkopaev, A., Koznov, D.: A Survey of Programming Language Memory Models. Programming and Computer Software **47**(6), 439–456 (2021)
10. Nelson, C.G.: Techniques for Program Verification. Ph.D. thesis, Stanford University (1980)
11. Nieuwenhuis, R., Oliveras, A.: Proof-Producing Congruence Closure. In: Term Rewriting and Applications. pp. 453–468. Springer, Berlin, Heidelberg (2005)
12. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. ACM SIGPLAN Notices **50**(6), 1–11 (2015)
13. Stone, J., Matsakis, N.: Rayon. https://crates.io/crates/rayon (2024), https://crates.io/crates/rayon, accessed 2024-06-07
14. Tarjan, R.E.: Efficiency of a Good But Not Linear Set Union Algorithm. Journal of the ACM **22**(2), 215–225 (1975)
15. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality Saturation: A New Approach to Optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 264–276. Association for Computing Machinery (2009)
16. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and Extensible Equality Saturation. Proceedings of the ACM on Programming Languages **5**(POPL), 1–29 (2021)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_49