Exploring benchmarking, profiling, and data parallelism in the egg project

Edmund Goodman (edjg2)

Word count: 2498

January 21, 2025

The core of the egg ("*e-graphs good*") project is a library providing a flexible and performant implementation of the equality graph (e-graph) data structure, providing a re-usable basis for leveraging equality saturation in program optimisers to address the phase-ordering problem of compilers. This mini-project explores the performance characteristics of the egg library through benchmarking and profiling techniques, and uses this information to guide the design and assess the suitability of data parallelism for its applications. We conclude that Rust's rich support for multithreading allows data parallelism to be leveraged in egg, but many portions of the algorithm rely on shared mutable state, so the approach can only provide incremental performance gains as a corollary of Amdahl's law.

Keywords: Equality saturation, e-graphs, performance profiling, data parallelism, rayon, Rust

1 Introduction

Optimising compilers have formed a key aspect of developing performant programs since the first FORTRAN compiler in 1954, which aimed to compete with average hand-coded assembly and dedicated 25% of its instructions to optimisations [1]. Since then, significant research and engineering effort has been exerted to build better optimising compilers, with notable examples including GCC [2] and the LLVM project [3], leveraging increasingly numerous and exotic optimisations to meet ever-increasing performance goals.

1.1 Phase ordering problem

The phase-ordering problem is a classical issue in compiler design, discussed as early as 1988 [4], referring to selecting the optimal order with which to apply optimisations such as term rewrites. Since the choice of earlier optimisations can hide or reveal later ones, this ordering can significantly impact the generated code and its performance characteristics [5].

A naïve approach to this issue entails applying all orderings of term rewrites and selecting the best by a cost metric. Unfortunately, this approach is exponential both space and time in the number of rewriting rules [6], which taken in combination with modern compilers using many substitutions (for example LLVM's documentation lists 53 transformation passes alone [7]) makes it intractable for real-world use. In addition to this, some substitutions such as $x \to x + 0$ are unbounded in their expansion, meaning the set of all orderings cannot be enumerated. As such, a more sophisticated approach is needed.

1.2 Equality graphs

The egg paper begins by introducing the equality graph (henceforth e-graph) data structure, which underpins their approach. E-graphs were first discussed in Nelson Gregory's PhD thesis in 1980 [8], and have later been defined in the literature by Joshi et al. as "a conventional term DAG augmented with an equivalence relation on the nodes of the DAG; two nodes are equivalent if the terms they

represent are identical in value" [9], who also note that an e-graph of size $O(n)$ can represent $\Theta(2^n)$ distinct rewrites – mitigating the exponential complexities discussed in §1.1.

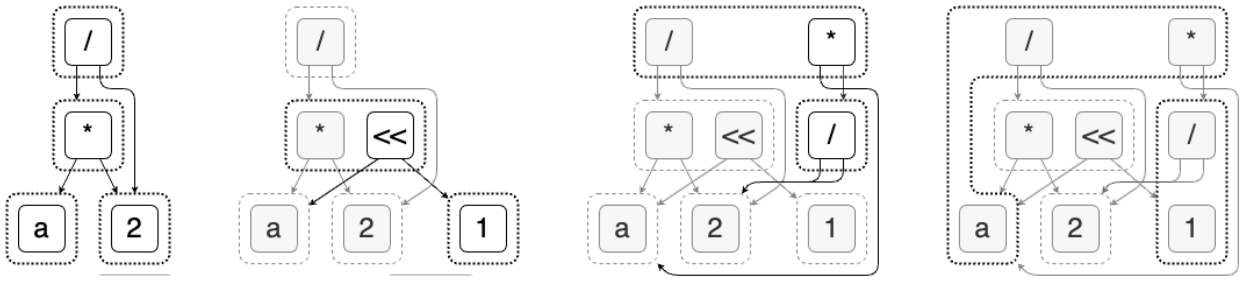
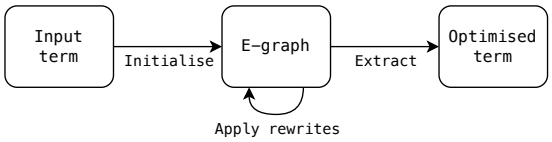


Figure 1: An e-graph compactly representing programs equivalent to $(a \times 2)/2$, from both the egg GitHub documentation [10] and the egg paper [6, Fig. 2]

The egg paper instead defines e-graphs more concretely in a way tied to their performant implementation, extending Galler and Fisher's seminal union-find data structure [11]. The disjoint sets of the structure (e-classes) represent equivalence classes of expressions (e-nodes), which are represented as functions whose arguments are other e-classes. An example of this is shown in Figure 1, constructing an e-graph left-to-right by applying term rewrites. In addition to this, the implementation maintains two invariants: congruence closure, meaning that if two expressions are equivalent, their sub-expressions are also equivalent; and Hashcons, maintaining canonical values in an internal data structure.

1.3 Equality saturation

Equality saturation is a novel approach to structuring the optimisation phase of a compiler, proposed by Tate et al. in 2009 [12]. It leverages the e-graph data structure to encode many equivalent expressions of a term under a set of optimisation rules, then extract the most optimal term from the e-graph according to a cost model, following the workflow shown in Figure 2. As opposed to naïve term rewriting where a poor early choice of optimisation ordering may destructively hide further optimisations, forming the crux of the phase ordering problem, equality saturation's constructive approach does not clobber later optimisations, resulting in more desirable time and space complexity characteristics. Despite the effectiveness of equality saturation, its widespread adoption was hindered by the combination of its complexity and the absence of a reference implementation which could be re-used.




```
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)
  while not egraph.is_saturated_or_timeout():
  for rw in rewrites:
    for (subst, eclass) in
    ↪  egraph.ematch(rw.lhs):
      eclass2 = egraph.add(rw.rhs.subst(subst))
      egraph.merge(eclass, eclass2)
  return egraph.extract_best()
```

Figure 2: Box diagram and pseudocode for the equality saturation approach to term optimisation, derived from [6, Fig. 3].

1.4 egg project

The egg project is significant in its research domain as it moved forward both the abstract and concrete capabilities of the equality saturation technique.

For the abstract capabilities, the paper's contributions include a novel optimisation to the way the e-graph invariants are maintained. In traditional approaches, reading and writing to the e-graph is interleaved for each rewrite, meaning the invariants must be maintained after each write. Instead, egg groups these phases for each rewrite separately, meaning the expensive invariant maintenance need only be performed once at the end of each iteration, rather than for every rewrite – shown in Listing 3.

```python
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)
  while not egraph.is_saturated_or_timeout():
    # Reading and writing is mixed
    for rw in rewrites:
      for (subst, eclass) in
      ↪  egraph.ematch(rw.lhs):
        # In traditional equality saturation,
        # matches can be applied right away
        # because invariants are always kept
        eclass2 = egraph.add(
        ↪  rw.rhs.subst(subst))
        egraph.merge(eclass, eclass2)
        # Restore the invariants at each merge
        egraph.rebuild()
  return egraph.extract_best()
```

(a) Traditional approach.

```python
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)
  while not egraph.is_saturated_or_timeout():
    matches = []
    # Read-only phase, invariants are preserved
    for rw in rewrites:
      for (subst, eclass) in
      ↪  egraph.ematch(rw.lhs):
        matches.append((rw, subst, eclass))
    # Write-only phase, temporarily break
    ↪  invariants
    for (rw, subst, eclass) in matches:
      eclass2 = egraph.add(rw.rhs.subst(subst))
      egraph.merge(eclass, eclass2)
    # Restore the invariants once per iteration
    egraph.rebuild()

  return egraph.extract_best()
```

(b) egg approach.

Figure 3: Comparison of the original and modified algorithms, demonstrating egg's novel "phase-separation" contribution, derived from [6, Fig. 5].

For the concrete capabilities, the open-source library mitigated a key problem precluding the widespread adoption of e-graph approaches, the complex and involved process of developing a custom e-graph implementation for each use case. For example, one of the few existing use cases of e-graphs, the Herbie floating point accuracy optimiser [13], initially implemented its own e-graph in Racket, but later switched to using egg, resulting in a performance uplift and identification of bugs.

Due both to the improved viability and reduced barrier to entry facilitated by egg, there has since been increased interest in leveraging e-graphs for a wide variety of use cases [14]. In combination, these contributions significantly increased the prominence of the research domain, with this impact being confirmed by the publication winning a "Distinguished Paper" award from its venue, POPL 2021.

1.5 Contributions

Despite the clear advantages of the equality saturation approach leveraged by egg over naïve enumeration, it is still limited in the scale of data it can process, by default limiting the maximum number of e-nodes to $10,000$ [15] and execution time to reach saturation to 5 seconds [16]. This limitation motivates an exploration of how the performance of egg's implementation can be improved, in order for it to process larger scales of data and hence be more viable for real-world workloads. As such, this mini-project make the following three contributions:

1. An extension of `egg`'s benchmarking suite, including performance profiling, to empirically justify optimisations, discussed in §2 and §3.
2. An exploration of opportunities for data parallelism in `egg` to augment its performance on multicore systems, discussed in §4.
3. An evaluation of the performance characteristics of data parallelism in `egg`, discussed in §5.

2 Benchmarking

`egg` provides minimal but fine-grained benchmarks, albeit only for two workloads. As such, the first extension we make to the project is porting the existing tests into benchmarks in the `criterion` testing framework [17], with an example shown in §A.1. This addresses the issues with the existing suite, providing a much more varied and representative set of workloads, and leverages the benefits of frameworks such as cache warmups and statistical analysis across many experimental runs. Using these new benchmarks, we can then better quantify and profile the performance characteristics of `egg`, informing later optimisations.

2.1 Build configuration optimisation

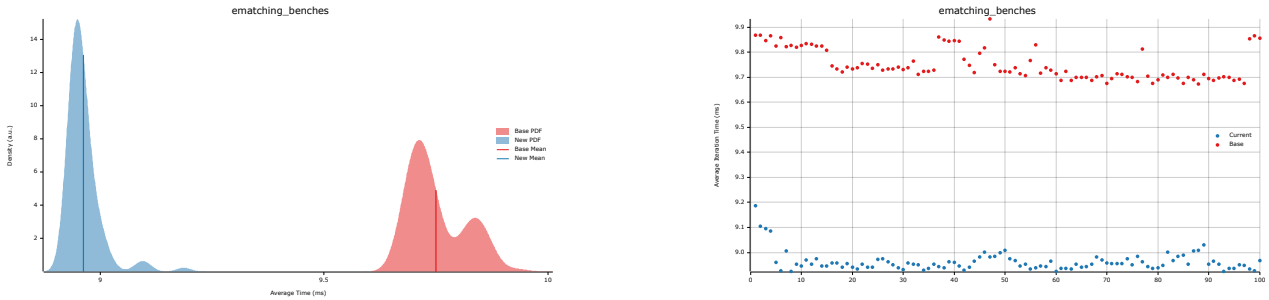
In many performance optimisation efforts, there are low-hanging fruit which can be easily modified with little consequence for non-negligible improvements. The most salient example in the `egg` codebase is modifications to the build configuration of the Rust compiler, shown in Listing 1. These modifications provide more information for better optimisation at compile time, albeit with a tradeoff for build times discussed in appendix §B.2. Whilst there are further opportunities for optimisation, the focus of this project is on data parallelism, so these are not explored for brevity.

```
[profile.release]
codegen-units = 1
lto = "fat"
panic = "abort"
```

Listing 1: Modifications to the build configuration in the `egg` `Cargo.toml` file.

2.2 Benchmarking evaluation

The benefits of the new benchmarks are demonstrated by measuring the impact of the build configuration changes across both suites. The existing benchmarks required manual re-runs and data processing shown in §B.1 to aggregate statistically significant results, with the `math` bench measured as having a 4.3% performance increase, albeit with significant variance. In contrast, the benefits of our new `criterion` benchmarks come into clear focus when running a similar experiment with only a single command, with Figure 4 showing the automatically generated experimental results. From these results, we can see that the modified build configuration (blue) is non-negligibly faster than the original build configuration (red), with a measured performance uplift of 8.1%.



(a) Probability distribution function of benchmark run times.

(b) Recorded times for each benchmark run in sequence.

Figure 4: Experimental results for running `egg`'s `math_ematching_bench` under our new `criterion` benchmark harness.

3 Performance profiling

Performance profiling extends beyond benchmarking, answering not only the question of what are the performance characteristics of a program, but also why does it have those characteristics. Often, this entails identifying the "worst offending" sections of a program which dominate its execution time. We leverage `samply` [18], a statistical profiler to instrument program runs, and visualise the results in two ways: flamegraphs and stack charts.

3.1 Understanding program performance with flamegraphs

Flamegraphs were developed by Brendan Gregg in 2011 to quickly identify the most frequent code paths during performance analysis [19]. They hierarchically aggregate the time spent in a program's call stack irrespective of temporal ordering, making those code paths most visually prominent. Figure 5 shows the flamegraph of an `egg` benchmark, which can then be used for identifying and characterising the performance of opportunities for data parallelism, discussed in §4.

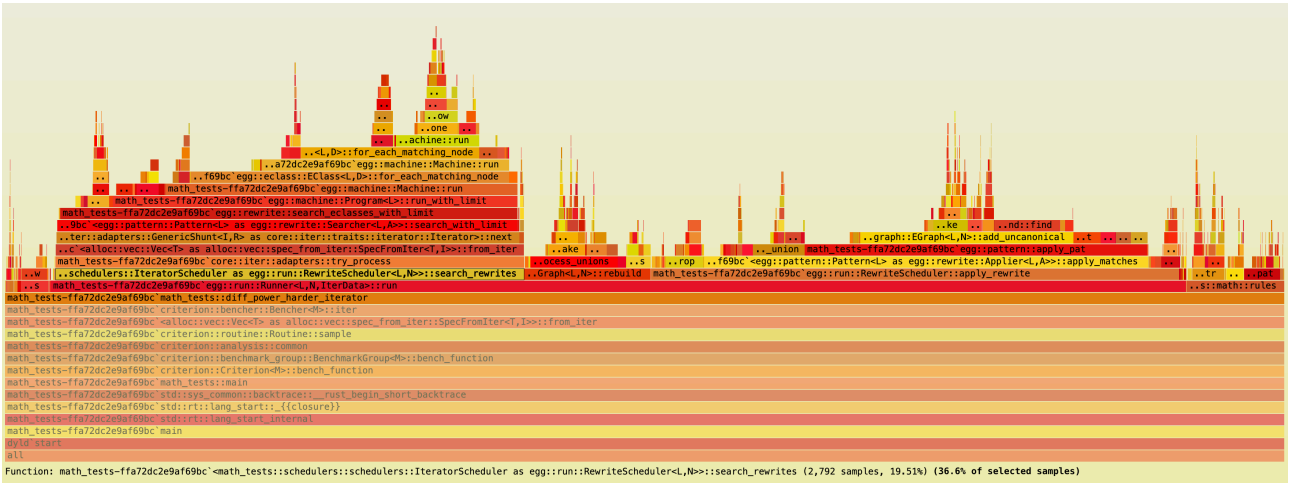
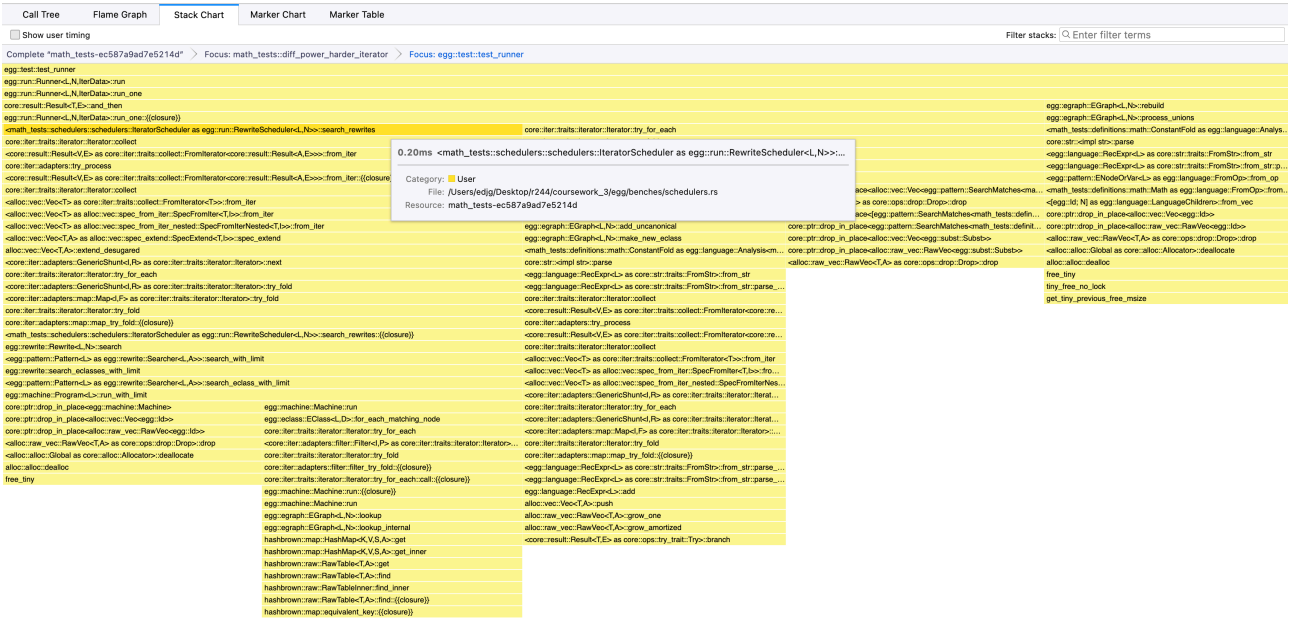


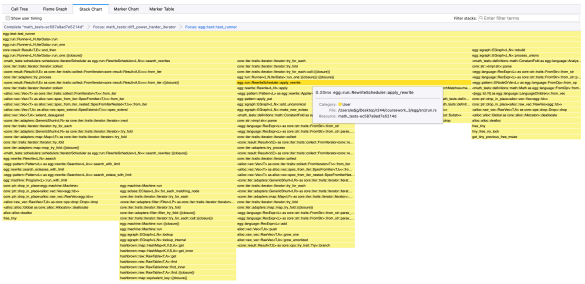
Figure 5: Flamegraph of the `diff_power_harder` benchmark, visualised with `cargo flamegraph` [19], full page version in §13.

3.2 Visualising phase separation with stack charts

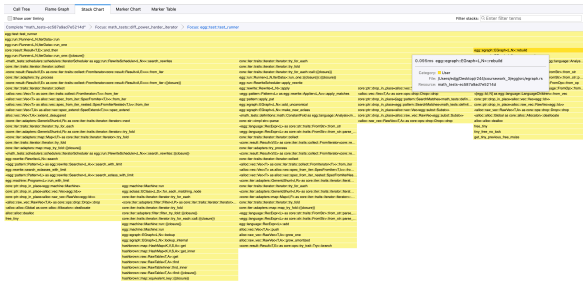
In contrast to flamegraphs, stack charts visualise a program's call stack whilst preserving temporal ordering, making them more suitable for understanding program behaviour. Using this visualisation of profiling data, we can characterise `egg`'s novel contribution of phase separation, as shown in Figure 6, which further empirically supports our understanding of performance characteristics.



(a) Profile trace with first phase, `search_rewrites`, highlighted.



(b) Profile trace with second phase, `apply_rewrite`, highlighted.



(c) Profile trace with third phase, `rebuild`, highlighted.

Figure 6: Profile traces for a single iteration of `egg`'s equality saturation, empirically measuring the novel phase separation approach sketched in Listing 3 and visualised with the Firefox profiler [20].

4 Opportunities for data parallelism

For many years, Dennard scaling yielded predictable year-on-year improvements to single-core execution [21], as clock frequencies could be increased without proportionally increasing power consumption as transistor sizes shrank. The breakdown of Dennard scaling from 2005 resulted in a stagnation in single-core execution performance, and consequently the exploration of multi-core processors to meet performance goals [22]. To leverage this new architectural trend, programs had to be designed to use components which could run in parallel.

One of Rust's key benefits is its support for "Fearless Concurrency" [23], with its borrow checker validating both memory and thread safety through the same ownership mechanism. Leveraging this

benefit, the `rayon` crate [24] provides a simple abstraction for data parallelism over Rust's iterator construct, allowing concurrent processing of data sequences, whilst prohibiting data races through compile-time checking. An example of this is shown for the dot product kernel in Listing 7.

```rust
pub fn dot_product(
    lhs: &[f64], rhs: &[f64]
) -> f64 {
    lhs.iter()
        .zip(rhs.iter())
        .map(|(x, y)| x * y)
        .sum()
}
```

(a) Iterator implementation.

```rust
use rayon::prelude::*;

pub fn dot_product(
    lhs: &[f64], rhs: &[f64]
) -> f64 {
    lhs.par_iter()
        .zip(rhs.par_iter())
        .map(|(x, y)| x * y)
        .sum()
}
```

(b) `rayon` implementation.

Figure 7: Serial and parallel implementations of a dot product kernel in Rust, showing the simplicity of `rayon`'s abstraction – merely changing `.iter()` to `.par_iter()`.

To identify which components are most profitable to parallelise, we examine the profiler flamegraph shown in Figure 5. Of the components which can be parallelised, the one which constitutes the largest proportion of program runtime should be selected, as a corollary of Amdahl's law [25]. Taking this approach, we can see that the `egg::Run::RewriteSchedule::search_rewrites` function in Figure 5 constitutes 36.6% of profiler samples and hence is the best candidate for parallelisation. This matches the unexplored opportunities for parallelism suggested by the paper, which states "searching for rewrite matches, which is the bulk of running time, can be parallelized thanks to the phase separation" [6, p. 23:20].

The `egg` project is structured to maximise reusability by allowing custom implementations of many components, including the algorithm for searching rewrites, by the developer of the dependent application. As such, to best represent such use cases, we provide an `egg::Run::RewriteSchedule` implementation for data parallelism over the identified component, show in Listing 2. This defines the mechanism by which the e-graph is searched for possible rewrites during equality saturation, one of the bottlenecks identified by profiling.

```rust
use rayon::prelude::*;

#[derive(Debug)]
pub struct ParallelIteratorScheduler;

impl<L, N> RewriteScheduler<L, N> for ParallelIteratorScheduler
where
    L: Language + Sync + Send,
    L::Discriminant: Sync + Send,
    N: Analysis<L> + Sync + Send,
    N::Data: Sync + Send
{
    fn search_rewrites<'a>(
        &mut self,
        iteration: usize,
```

```
16          egraph: &EGraph<L, N>,
17          rewrites: &[&'a Rewrite<L, N>],
18          limits: &RunnerLimits,
19      ) -> RunnerResult<Vec<Vec<SearchMatches<'a, L>>>> {
20          rewrites
21              .par_iter()
22              .map(|rw| {
23                  let ms = rw.search(egraph);
24                  limits.check_limits(iteration, egraph)?;
25                  Ok(ms)
26              })
27              .collect()
28      }
29  }
30
```

Listing 2: Rust implementation of a parallel rewrite scheduler using `rayon`, directly usable in `egg` dependent runners with `.with_scheduler(ParallelIteratorScheduler)`.

Unfortunately, opportunities for data parallelism are constrained by task dependencies and shared mutable state. In order to guarantee to the Rust compiler these conditions are met, we set the `Sync` and `Send` traits on lines 8 – 11 of Listing 2. However, these are not valid for the default `egg::Run::BackoffScheduler` implementation, as it relies heavily on shared mutable state to track which rules to ban, so it cannot trivially be parallelised. Hence, we evaluate performance characteristics of a simple iterator scheduler, which is then parallelised as shown above.

5 Evaluation

In order to characterise the performance of the implementation of data parallelism discussed in §4, we modify our benchmark suite from §2 to allow direct comparison between serial and parallel scheduled runners – the main functional loop of `egg`'s equality saturation implementation. For the following plots, we discuss the `diff_power_harder` benchmark from the variety investigated for its comparative complexity among tests to minimise the impact of setup and teardown, making it better representative of real-world applications of equality saturation.

Figure 8, with individual plots in §D.1, directly compares the average runtime using `criterion`. From this, we can see that the parallel implementation is $\sim 12\%$ slower than the serial implementation.

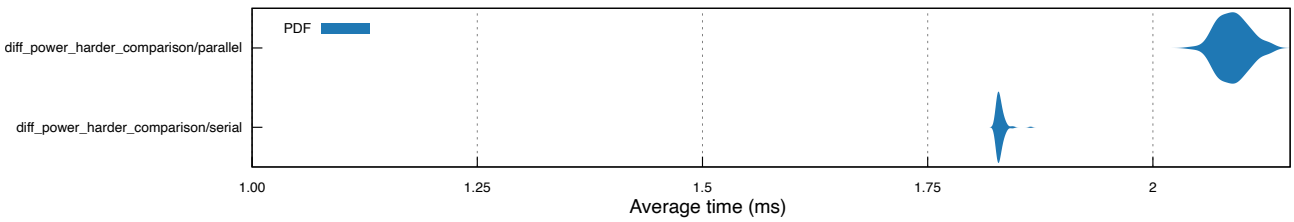
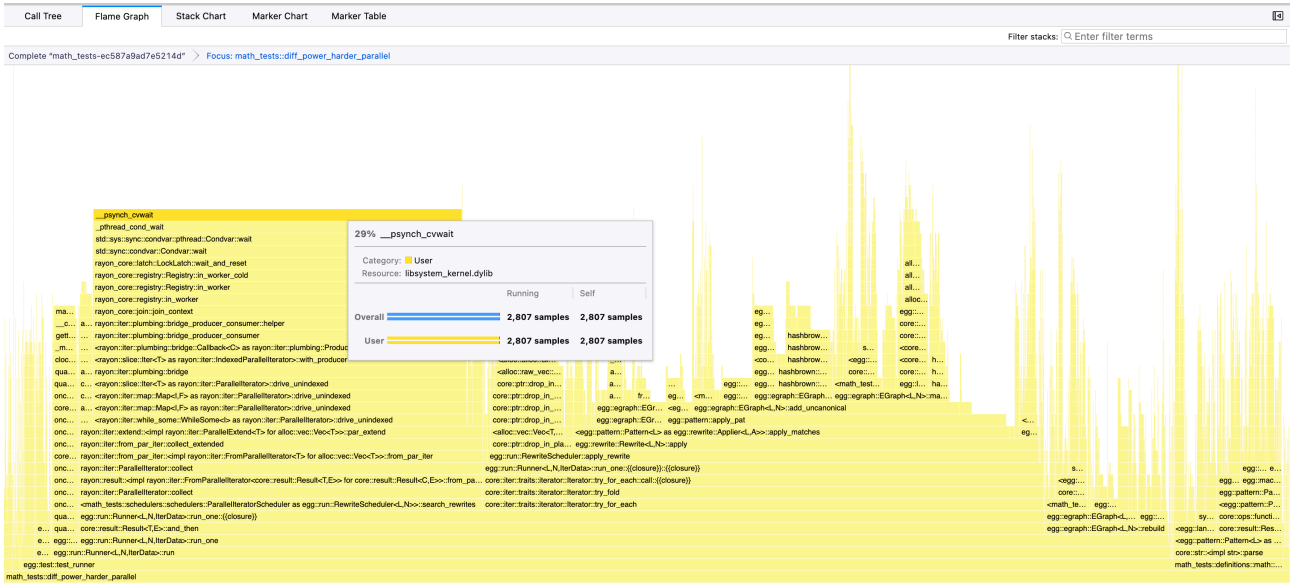


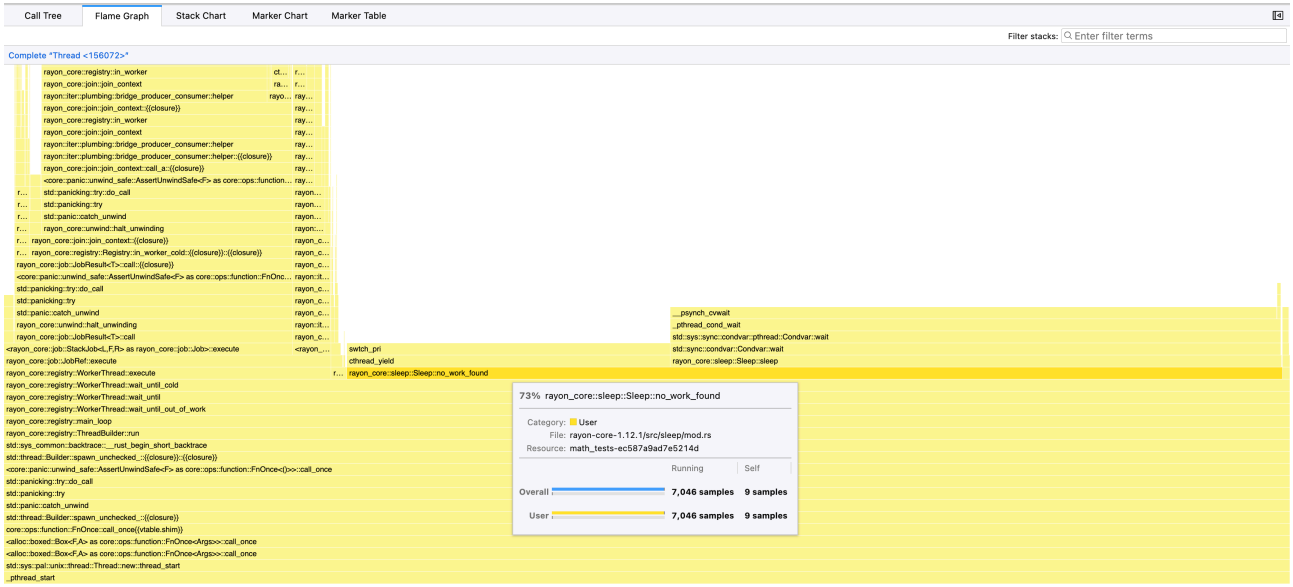
Figure 8: Comparison of the average runtimes of the serial and parallel implementations of the `diff_power_harder` benchmark, generated using `criterion`.

This regression could be caused by a number of issues, from overhead spawning threads to long waits for locks on shared mutable state. As such, we again apply the performance profiling techniques

discussed in §3, profiling the program run and visualising the data with the Flamegraphs shown in Figure 9.



(a) Main thread.



(b) Worker thread.

Figure 9: Flamegraphs of the parallel implementation of the `diff_power_harder` benchmark's main and worker threads, visualised with the Firefox profiler [20].

In Figure 9a, the highlighted vertical block culminating in `__psynch_wait` shows where the `egg::Run::RewriteSchedule::search_rewrites` is dispatched to the worker threads, and Figure 9b shows From this, we can see that the vast majority, 73%, of the worker thread time is spent waiting as opposed to searching rewrites. Since Rayon uses a work-stealing scheduled thread model [24], if the number of rewrites to search is fewer than the large default thread pool size, which may be guaranteed for small rule sets, some threads may do no work whilst still incurring a scheduling overhead. This can be mitigated by reducing the number of threads in the thread pool with the `RAYON_NUM_THREADS` environment variable. Re-running the `diff_power_harder` tests shown in Figure 10 with 2 threads then shows a performance uplift of $\sim 10\%$ by leveraging data parallelism.

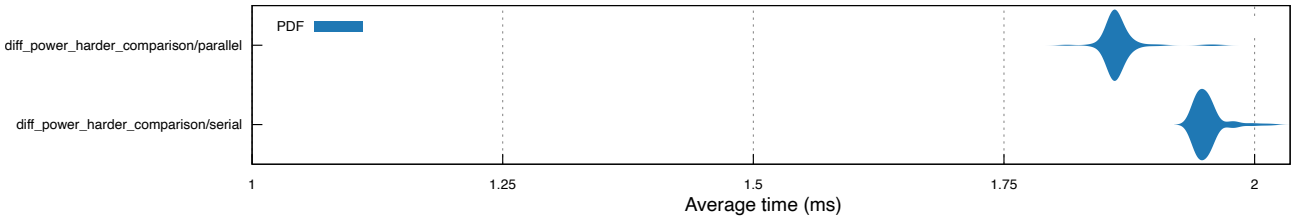


Figure 10: Comparison of the average runtimes of the serial and parallel implementation using only two threads of the `diff_power_harder` benchmark, generated using `criterion`.

To characterise this behaviour, we measure the performance results sweeping through the size of the thread pool in comparison to serial runs, shown in Figure 11. From this, we can see that data parallelism provides performance uplifts for the `diff_power_harder` workload for between 2 and 6 threads, with larger thread pool sizes incurring an overhead which offsets any benefits derived from parallelisation.

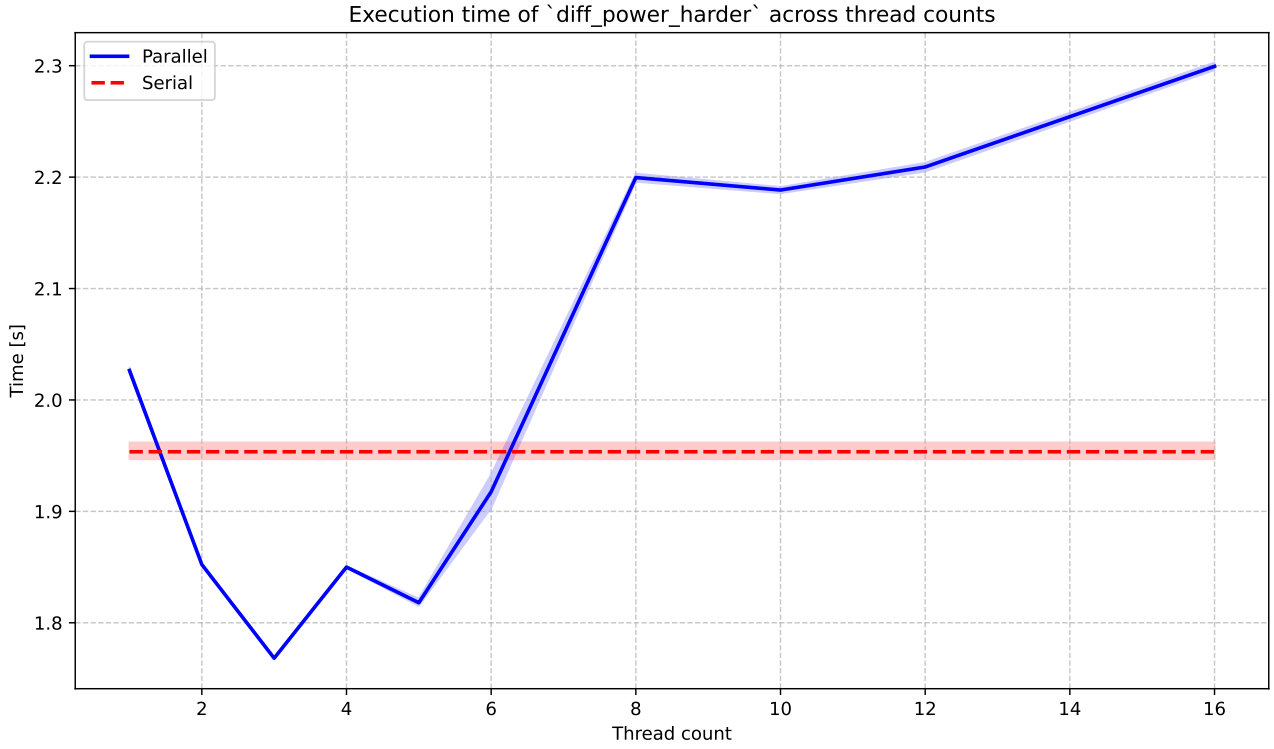


Figure 11: Plot of the execution time of the `diff_power_harder` test across thread pool sizes when run in parallel, and when run in serial.

5.1 Performance scaling

Finally, we explore the scaling properties of the parallel implementation, with the experiment shown in Figure 12. We can see that as workloads scale the performance uplift from parallelism commensurately grows as the threading overhead is amortised over the longer runtimes. However, for this the `egg::Run::RewriteSchedule::search_rewrites` represents a smaller proportion of the work, so the parallel version is less beneficial than previous experiments at $\sim 4\%$ performance uplift.

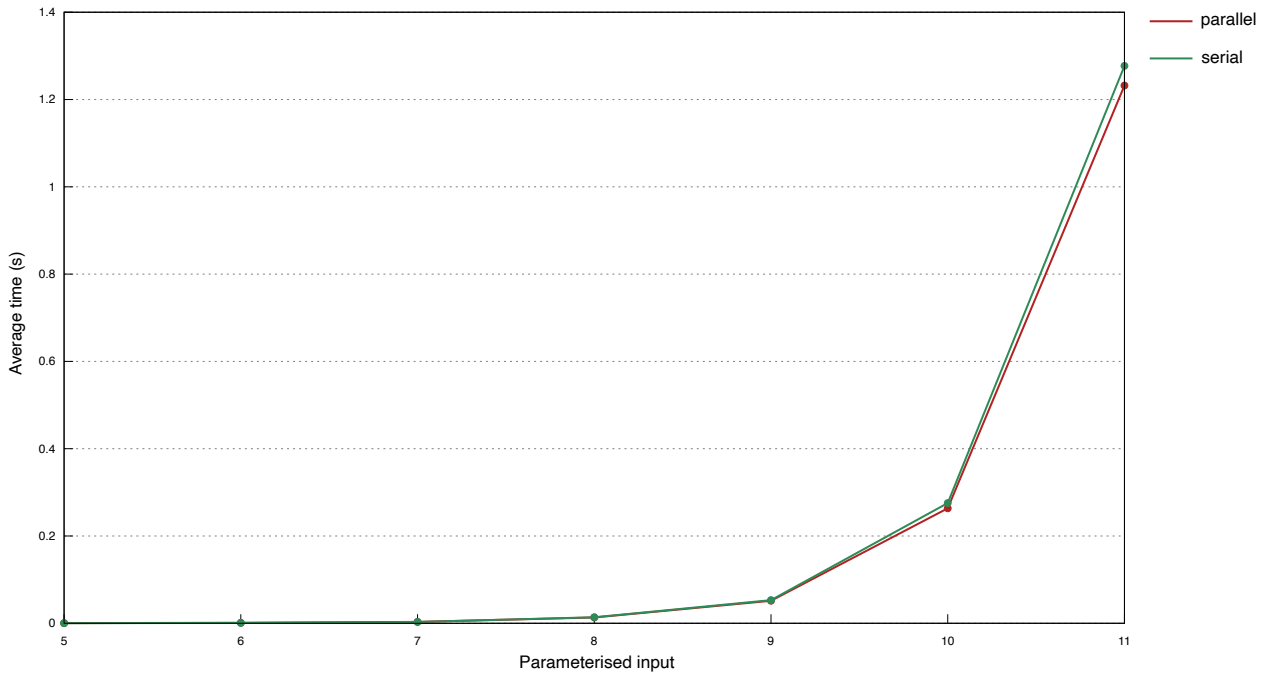


Figure 12: Comparison of the scaling properties of the serial and parallel implementations, over a parameterised version of the `math_associate_adds` test case shown in appendix §A.2.

6 Future and related work

Following its initial success, the `egg` project has as remained an active research topic. In addition to the rewrite of Herbie discussed in §1.4, it has underpinned a variety of developments in compiler design, for example Yang et al.'s use application of equality saturation for tensor graph superoptimisation [26]. In addition to this, two years after its publication, Zhang et al. proposed `egglog` [27], which unifies `egg`'s equality saturation with Datalog [28], a declarative programming language influenced by Prolog.

Whilst equality saturation provides an analytical approach to the tractability of the phase ordering problem, recent trends in reinforcement learning have also been applied to the problem, with examples including LLVM's POSET-RL [29] and Haj-Ali et al.'s AutoPhase [30] respectively, which both achieved over 10% performance improvements for their workloads.

7 Conclusion

To conclude, we extended `egg`'s benchmarking suite, and used it in combination with performance profiling techniques to implement and evaluate opportunities for data parallelism in `egg`. We found that searching the e-graph for rewrites is the most suitable application of parallelism, and can yield incremental performance improvements. However, these improvements are constrained by necessarily serial components of equality saturation, as a corollary of Amdahl's law.

8 References

[1] Paul B. Schneck. "A Survey of Compiler Optimization Techniques". In: *Proceedings of the ACM Annual Conference*. ACM '73. New York, NY, USA: Association for Computing Machinery, Aug. 27, 1973, pp. 106–113. ISBN: 978-1-4503-7490-3. DOI: 10.1145/800192.805690. URL: https://dl.acm.org/doi/10.1145/800192.805690 (visited on 01/06/2025).

[2] Richard Stallman and GCC Team. *GCC, the GNU Compiler Collection - GNU Project*. URL: https://gcc.gnu.org/ (visited on 01/12/2025).

[3] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* International Symposium on Code Generation and Optimization, 2004. CGO 2004. Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. URL: https://ieeexplore.ieee.org/abstract/document/1281665 (visited on 11/18/2024).

[4] M. E. Benitez and J. W. Davidson. "A Portable Global Optimizer and Linker". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. New York, NY, USA: Association for Computing Machinery, June 1, 1988, pp. 329–338. ISBN: 978-0-89791-269-3. DOI: 10.1145/53990.54023. URL: https://dl.acm.org/doi/10.1145/53990.54023 (visited on 01/06/2025).

[5] Yu Wang, Hongyu Chen, and Ke Wang. *Beyond the Phase Ordering Problem: Finding the Globally Optimal Code w.r.t. Optimization Phases*. Oct. 22, 2024. DOI: 10.48550/arXiv.2410.03120. arXiv: 2410.03120 [cs]. URL: http://arxiv.org/abs/2410.03120 (visited on 01/06/2025). Pre-published.

[6] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, et al. "Egg: Fast and Extensible Equality Saturation". In: *Artifact for "Fast and Extensible Equality Saturation"* 5 (POPL Jan. 4, 2021), 23:1–23:29. DOI: 10.1145/3434304. URL: https://dl.acm.org/doi/10.1145/3434304 (visited on 11/22/2024).

[7] *LLVM's Analysis and Transform Passes — LLVM 20.0.0git Documentation*. URL: https://www.llvm.org/docs/Passes.html (visited on 01/06/2025).

[8] Nelson Charles Gregory. "Techniques for Program Verification". PhD thesis. Ph. D. Dissertation. Stanford University, United States-California. AAI8011683, 1980.

[9] Rajeev Joshi, Greg Nelson, and Keith Randall. "Denali: A Goal-Directed Superoptimizer". In: *SIGPLAN Not.* 37.5 (May 17, 2002), pp. 304–314. ISSN: 0362-1340. DOI: 10.1145/543552.512566. URL: https://dl.acm.org/doi/10.1145/543552.512566 (visited on 01/12/2025).

[10] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, et al. *Egraphs-Good/Egg*. egraphs-good, Jan. 8, 2025. URL: https://github.com/egraphs-good/egg (visited on 01/11/2025).

[11] Bernard A. Galler and Michael J. Fisher. "An Improved Equivalence Algorithm". In: *Commun. ACM* 7.5 (May 1, 1964), pp. 301–303. ISSN: 0001-0782. DOI: 10.1145/364099.364331. URL: https://dl.acm.org/doi/10.1145/364099.364331 (visited on 01/11/2025).

[12] Ross Tate, Michael Stepp, Zachary Tatlock, et al. "Equality Saturation: A New Approach to Optimization". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. New York, NY, USA: Association for Computing Machinery, Jan. 21, 2009, pp. 264–276. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480915. URL: https://dl.acm.org/doi/10.1145/1480881.1480915 (visited on 01/12/2025).

[13] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, et al. "Automatically Improving Accuracy for Floating Point Expressions". In: *SIGPLAN Not.* 50.6 (June 3, 2015), pp. 1–11. ISSN: 0362-1340. DOI: 10.1145/2813885.2737959. URL: https://dl.acm.org/doi/10.1145/2813885.2737959 (visited on 01/11/2025).

[14] *EGRAPHS Community Meeting*. URL: https://egraphs.org/meeting/ (visited on 01/11/2025).

[15] Max Willsey. *Egg/Src/Run.Rs L343 at 2750f55 · Egraphs-Good/Egg*. GitHub. URL: https://github.com/egraphs-good/egg/blob/2750f55eac565c1fd14c1d3cb72d84851ddf7460/src/run.rs#L343 (visited on 01/11/2025).

[16] Max Willsey. *Egg/Src/Run.Rs L344 at 2750f55 · Egraphs-Good/Egg*. GitHub. URL: https://github.com/egraphs-good/egg/blob/2750f55eac565c1fd14c1d3cb72d84851ddf7460/src/run.rs#L344 (visited on 01/11/2025).

[17] Brook Heisler. *Bheisler/Criterion.Rs*. Jan. 13, 2025. URL: https://github.com/bheisler/criterion.rs (visited on 01/13/2025).

[18] Markus Stange. *Mstange/Samply*. Jan. 19, 2025. URL: https://github.com/mstange/samply (visited on 01/19/2025).

[19] Brendan Gregg. *Brendangregg/FlameGraph*. Dec. 3, 2024. URL: https://github.com/brendangregg/FlameGraph (visited on 12/04/2024).

[20] *Firefox-Devtools/Profiler*. Firefox DevTools, Jan. 19, 2025. URL: https://github.com/firefox-devtools/profiler (visited on 01/19/2025).

[21] Mark Bohr. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper". In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 11–13. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2007.4785534. URL: https://ieeexplore.ieee.org/abstract/document/4785534 (visited on 11/02/2024).

[22] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, et al. "Dark Silicon and the End of Multicore Scaling". In: *IEEE Micro* 32.3 (May 2012), pp. 122–134. ISSN: 1937-4143. DOI: 10.1109/MM.2012.17. URL: https://ieeexplore.ieee.org/document/6175879 (visited on 10/14/2024).

[23] Steve Klabnik and Carol Nichols. "Fearless Concurrency". In: *The Rust Programming Language*. No Starch Press, 2023. ISBN: 978-1-71850-310-6.

[24] *Rayon-Rs/Rayon*. rayon-rs, Dec. 4, 2024. URL: https://github.com/rayon-rs/rayon (visited on 12/04/2024).

[25] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, Apr. 18, 1967, pp. 483–485. ISBN: 978-1-4503-7895-6. DOI: 10.1145/1465482.1465560. URL: https://dl.acm.org/doi/10.1145/1465482.1465560 (visited on 01/13/2025).

[26] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, et al. "Equality Saturation for Tensor Graph Superoptimization". In: *Proceedings of Machine Learning and Systems* 3 (Mar. 15, 2021), pp. 255–268. URL: https://proceedings.mlsys.org/paper_files/paper/2021/hash/cc427d934a7f6c0663e5923f49eba531-Abstract.html (visited on 01/13/2025).

[27] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, et al. *Better Together: Unifying Datalog and Equality Saturation*. May 15, 2023. DOI: 10.48550/arXiv.2304.04332. arXiv: 2304.04332. URL: http://arxiv.org/abs/2304.04332 (visited on 11/22/2024). Pre-published.

[28] David Maier, K. Tuncay Tekle, Michael Kifer, et al. "Datalog: Concepts, History, and Outlook". In: *Declarative Logic Programming: Theory, Systems, and Applications*. Vol. 20. Association for Computing Machinery and Morgan & Claypool, Sept. 1, 2018, pp. 3–100. ISBN: 978-1-970001-99-0. URL: https://doi.org/10.1145/3191315.3191317 (visited on 01/19/2025).

[29] Shalini Jain, Yashas Andaluri, S. VenkataKeerthy, et al. "POSET-RL: Phase Ordering for Optimizing Size and Execution Time Using Reinforcement Learning". In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). May 2022, pp. 121–131. DOI: 10.1109/ISPASS55109.2022.00012. URL: https://ieeexplore.ieee.org/abstract/document/9804673 (visited on 01/20/2025).

[30] Ameer Haj-Ali, Qijing Huang, William Moses, et al. *AutoPhase: Compiler Phase-Ordering for High Level Synthesis with Deep Reinforcement Learning*. Version 1. Jan. 15, 2019. DOI: 10.48550/arXiv.1901.04615. arXiv: 1901.04615 [cs]. URL: http://arxiv.org/abs/1901.04615 (visited on 01/20/2025). Pre-published.

A Example `criterion` benchmarks

A.1 Simple benchmark

Benchmarks derived from all the test cases for the `math`, `lambda`, and `simple` rule sets were written to ensure representation of real-world use cases. An example benchmark taken from the `math` suite is shown in Listing 3.

```rust
use egg::{rewrite as rw, *};

mod definitions;
use definitions::math;

use criterion::{criterion_group, criterion_main, Criterion, BenchmarkId};

fn math_simplify_add() {
    egg::test::test_runner(
        "math_simplify_add",
        None,
        &math::rules(),
        "(+ x (+ x (+ x x)))".parse().unwrap(),
        &["(* 4 x)".parse().unwrap()],
        None,
        true
    )
}

pub fn math_test_simplify_add(c: &mut Criterion) {
    c.bench_function(
        "math_simplify_add",
        |b| b.iter(math_simplify_add)
    );
}

criterion_group!(benches, math_test_simplify_add);
criterion_main!(benches);
```

Listing 3: Example benchmark drawn from the `math` test suite.

A.2 Parameterised benchmark

In addition to this, some benchmarks were modified to be parameterised in the size of their input data to investigate scaling properties. One example of this is the `math_associate_adds` test, shown in Listing 3.

```rust
use egg::*;

mod definitions;
use definitions::math;

use criterion::{criterion_group, criterion_main, Criterion, BenchmarkId};
```

```rust

fn generate_ascending_rpn(n: usize) -> String {
    if n == 0 {
        return String::new();
    }
    if n == 1 {
        return "1".to_string();
    }
    let mut result = format!("{}", n);
    for i in (1..n).rev() {
        result = format!("(+ {} {})", i, result);
    }
    result
}

fn generate_descending_rpn(n: usize) -> String {
    if n == 0 {
        return String::new();
    }
    if n == 1 {
        return "1".to_string();
    }
    let mut result = "1".to_string();
    for i in 2..=n {
        result = format!("(+ {} {})", i, result);
    }
    result
}

fn math_associate_adds(n: usize) {
    egg::test::test_runner(
        "math_associate_adds",
        Some(Runner::default()
                .with_time_limit(std::time::Duration::from_secs(120))
                .with_iter_limit(60)
                .with_node_limit(50_000_000)),
        &[
            rw!("comm-add"; "(+ ?a ?b)" ⇒ "(+ ?b ?a)"),
            rw!("assoc-add"; "(+ ?a (+ ?b ?c))" ⇒ "(+ (+ ?a ?b) ?c)"),
        ],
        generate_ascending_rpn(n).parse().unwrap(),
        &[generate_descending_rpn(n).parse().unwrap()],
        Some(|_: Runner<math::Math, ()>| ()),
        true
    )
}

pub fn math_scaling(c: &mut Criterion) {
    let mut group = c.benchmark_group("math_scaling");
```

```
    group.sample_size(10); // Bound the number of samples to avoid overwhelming
    ↪ profiler
    for i in 3..12 {
        group.bench_with_input(BenchmarkId::new("math_associate_adds", i), &i,
        |b, i| b.iter(|| math_associate_adds(*i)));
    }
    group.finish();
}

criterion_group!(benches, math_scaling);
criterion_main!(benches);
```

Listing 4: Example parameterised `math_associate_adds` number benchmark.

A.3 Fibonacci benchmark

Another example of a parameterised benchmark is the `lambda_fib` test, also shown in Listing 5.

```
use egg::*;

mod definitions;
use definitions::lambda;

use criterion::{criterion_group, criterion_main, Criterion, BenchmarkId};

fn fibonacci(n: u32) -> u32 {
    if n <= 1 {
        return n;
    }
    let mut a = 0;
    let mut b = 1;
    for _ in 2..=n {
        let temp = a + b;
        a = b;
        b = temp;
    }
    b
}

fn lambda_fib(n: u32) {
     egg::test::test_runner(
        "lambda_fib",
         Some(Runner::default()
            .with_time_limit(std::time::Duration::from_secs(60))
            .with_node_limit(25_000_000)
            .with_iter_limit(60)),
            &lambda::rules(),
            format!("(let fib (fix fib (lam n
            (if (= (var n) 0)
                0
            (if (= (var n) 1)
```

```
                1
            (+ (app (var fib)
                    (+ (var n) -1))
               (app (var fib)
                    (+ (var n) -2)))))))
            (app (var fib) {n}))").parse().unwrap(),
            &[fibonacci(n).to_string().parse().unwrap()],
        None,
        true
    );
}

pub fn lambda_fib(c: &mut Criterion) {
    let mut group = c.benchmark_group("lambda_test");
    group.sample_size(10); // Bound the number of samples to avoid overwhelming
    ↪ profiler
    for i in 2..6 {
        group.bench_with_input(BenchmarkId::new("fib", i), &i,
        |b, i| b.iter(|| lambda_fib(*i)));
    }
    group.finish();
}

criterion_group!(benches, lambda_fib);
criterion_main!(benches);
```

Listing 5: Example parameterised Fibonacci number benchmark.

When instrumented with the `peak_alloc` crate and evaluated with the parameter set to six, this benchmark shown in Listing 5 yields the logs shown in Listing 6.

```
[snip]
[INFO  egg::run] Current allocated memory: 1257.3353MB
[INFO  egg::run] Current allocated memory: 1257.3206MB
[INFO  egg::run] Current allocated memory: 1257.3057MB
[INFO  egg::run] Apply time: 10.39106763
[INFO  egg::run] Rebuild time: 5.21461168
[INFO  egg::run] Size: n=8625732, e=35256
[INFO  egg::run]
    Iteration 16
[INFO  egg::run] Search time: 8.539660830999999
[INFO  egg::run] Current allocated memory: 17535.623MB
[INFO  egg::run] Current allocated memory: 17535.576MB
[INFO  egg::run] Current allocated memory: 17535.572MB
[INFO  egg::run] Current allocated memory: 17502.984MB
^Cmake: *** [bench] Interrupt: 2
```

Listing 6: Logs from evaluating `fib(6)`.

B Build configuration changes

B.1 Existing benchmark experimental results

To measure the effect of the build configuration changes with the existing benchmark suite, we must calculate the differences between the output CSVs generated before and after the runs, across five re-runs for statistical confidence. This manually aggregated is shown below in Table 1.

Table 1: Comparison of the original and modified build configuration for the `math` tests under `egg`'s original benchmark infrastructure.

Test name	Avg default [s]	Avg configured [s]	Difference [%]
`diff_power_harder`	0.002738327	0.002623199	4.204318914
`diff_power_simple`	0.000345321	0.000326406	5.477512228
`integ_one`	0.000016351	0.000015018	8.152406581
`integ_part1`	0.000576379	0.000547154	5.070448438
`integ_part2`	0.002190299	0.002057403	6.067482111
`integ_part3`	0.000243557	0.00023004	5.549830225
`integ_sin`	0.000015725	0.000016792	-6.785373609
`integ_x`	0.000027	2.6267E-05	2.714814815
`math_associate_adds`	0.002685307	0.002411284	10.20453155
`math_diff_different`	0.000015559	0.000019977	-28.39514108
`math_diff_ln`	0.000017667	0.000015833	10.38093621
`math_diff_same`	1.8875E-05	0.000014351	23.96821192
`math_diff_simple1`	0.000236122	0.000220601	6.573296855
`math_diff_simple2`	0.000172896	0.000163921	5.190981862
`math_powers`	0.000024267	0.000025018	-3.09473771
`math_simplify_add`	0.000139948	0.00013663	2.370880613
`math_simplify_const`	0.000064576	0.000059435	7.961162042
`math_simplify_factor`	0.000544234	0.000492827	9.445753114
`math_simplify_root`	0.000793088	0.000733844	7.470041156

Positive percentage differences indicate that the modified version is faster, with `math_diff_different` and `math_diff_same` being excluded as outliers due to machine noise. Experimental data is generated using the command provided in the documentation, shown in Listing 7 for both build configurations, then compared with the percentage difference calculated as $\frac{d-c}{(d+c)\div 2} \times 100\%$.

```
EGG_BENCH_CSV=math.csv cargo test --test math --release -- --nocapture --test
↪  --test-threads=1
```

Listing 7: Command to generate the benchmark data.

B.2 Build configuration change compilation times

The compilation times can then be recorded using the `hyperfine` tool as shown in Listing 8, with the results shown in Table 2.

```
hyperfine --runs 10 "cargo clean && cargo build --profile release"
hyperfine --runs 5 "cargo clean && cargo test --release --no-run"
```

Listing 8: Command to generate the benchmark data building the library only and both the library and the tests respectively.

Table 2: Comparison of the original and modified build configuration compilation times.

Build configuration	Components built	Average build time [s]	Standard deviation [s]
Original	Library	3.426	0.042
Modified	Library	3.139	0.031
Original	Library and tests	8.292	0.042
Modified	Library and tests	16.582	1.377

From this table, we can see that modified build configuration has very little effect on building the library only. However, building the library with all the tests doubles the build time. This is because the tests define many rule sets, and include many macro expansions to set up test runners. Since dependent applications are likely to invoke fewer macro expansions they are unlikely to experience such a significant, and as applications are typically run many more times than they are compiled, the performance trade-off is likely still worthwhile.

C Full size flamegraph

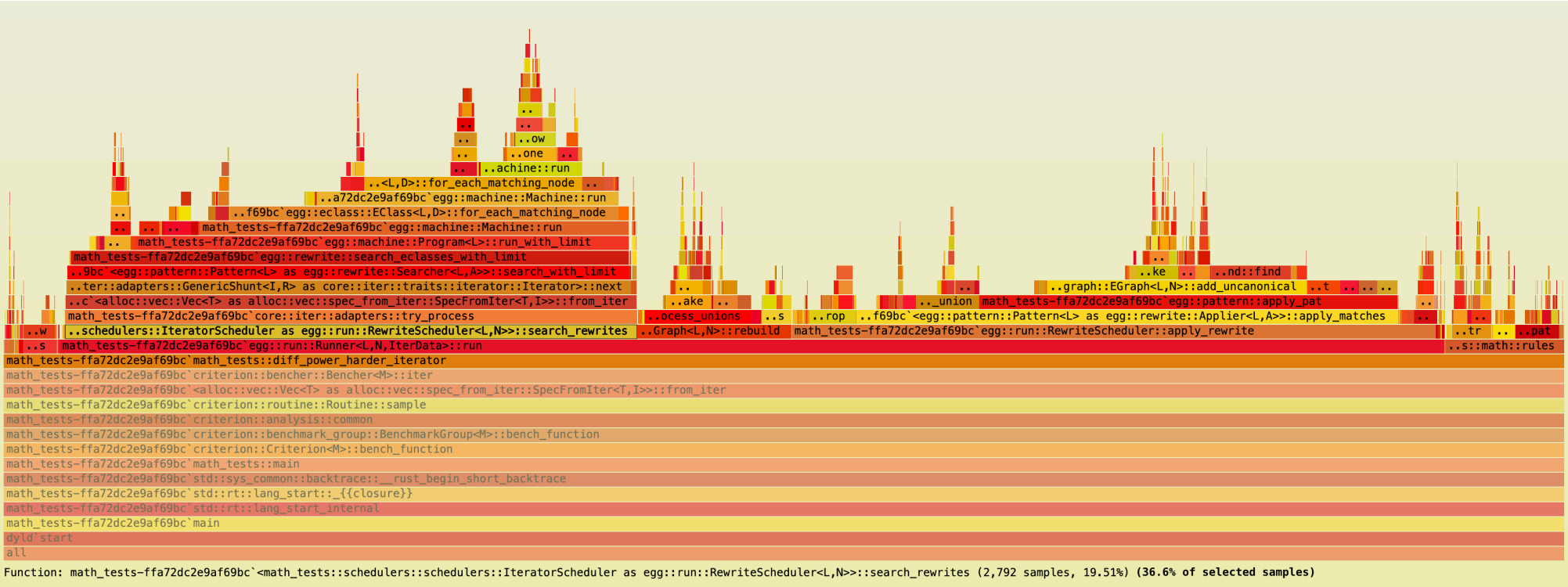


Figure 13: Full size flamegraph of the `diff_power_harder` benchmark, visualised with `cargo flamegraph` [19].

D Parallel implementation individual results

D.1 Results without thread pool size set

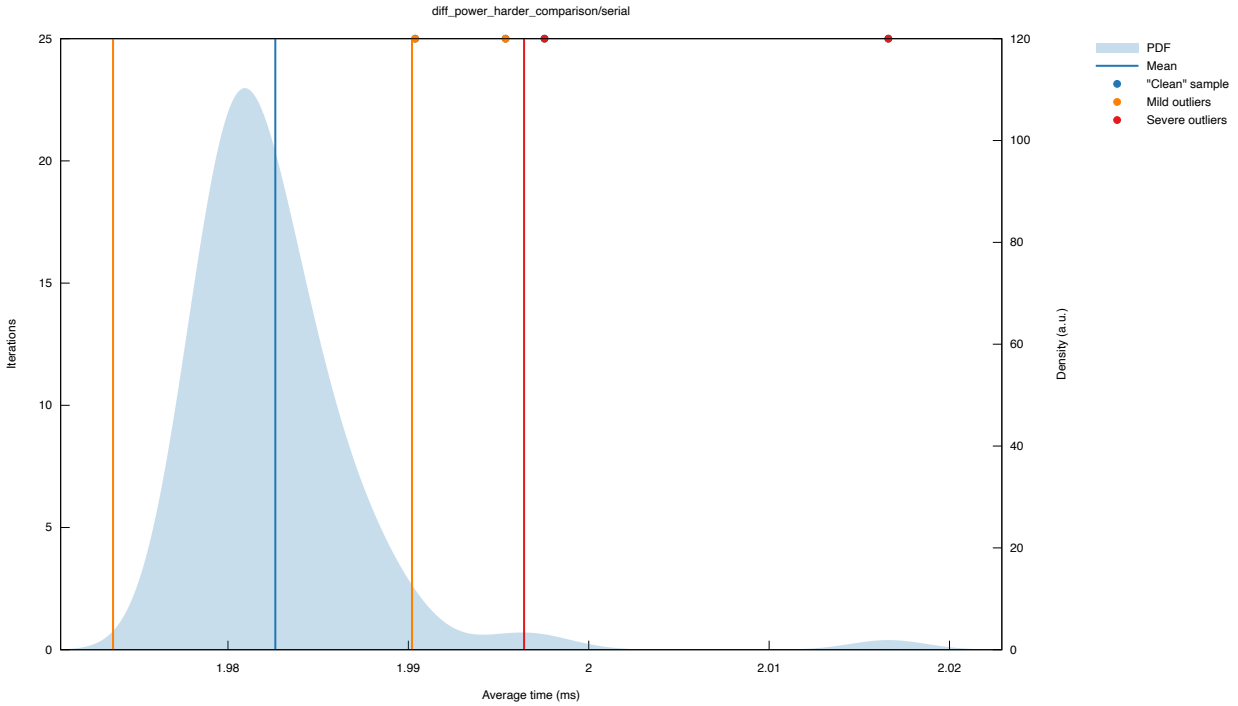


Figure 14: Runtime distribution of the serial execution of the `diff_power_harder` benchmark, generated by `criterion`.

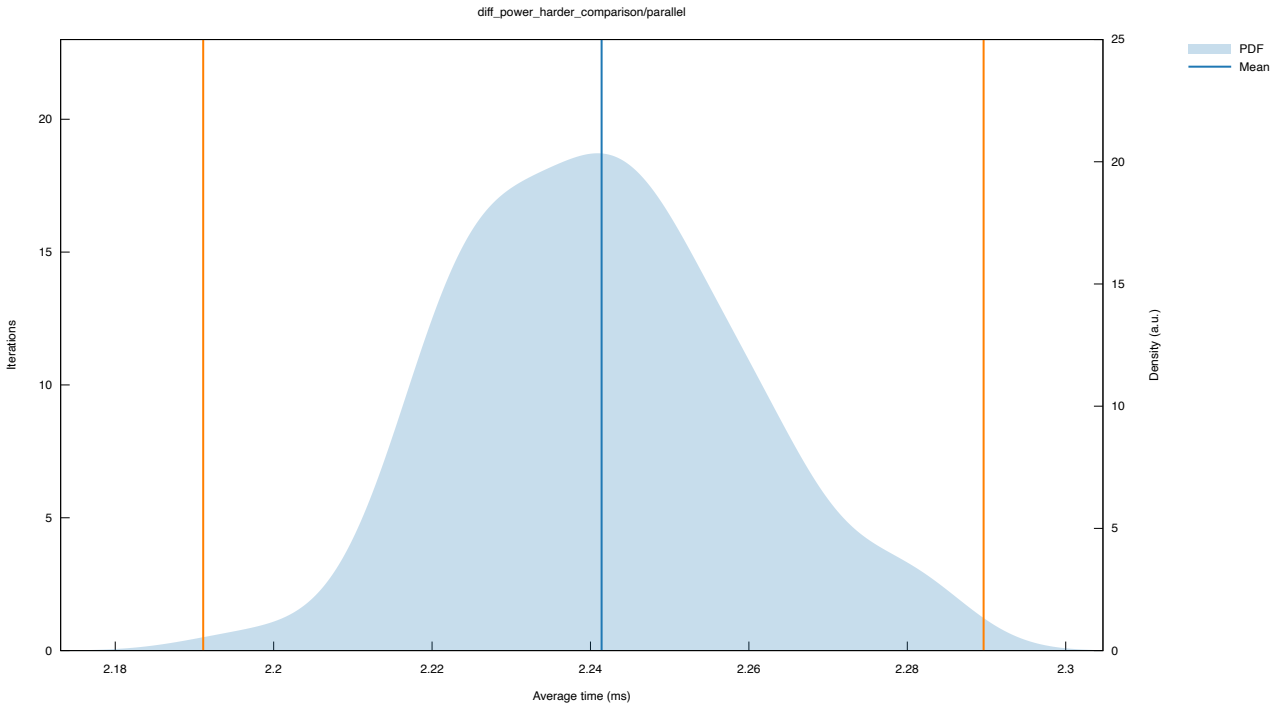


Figure 15: Runtime distribution of the parallel execution of the `diff_power_harder` benchmark, generated by `criterion`.

D.2 Results with thread pool size set

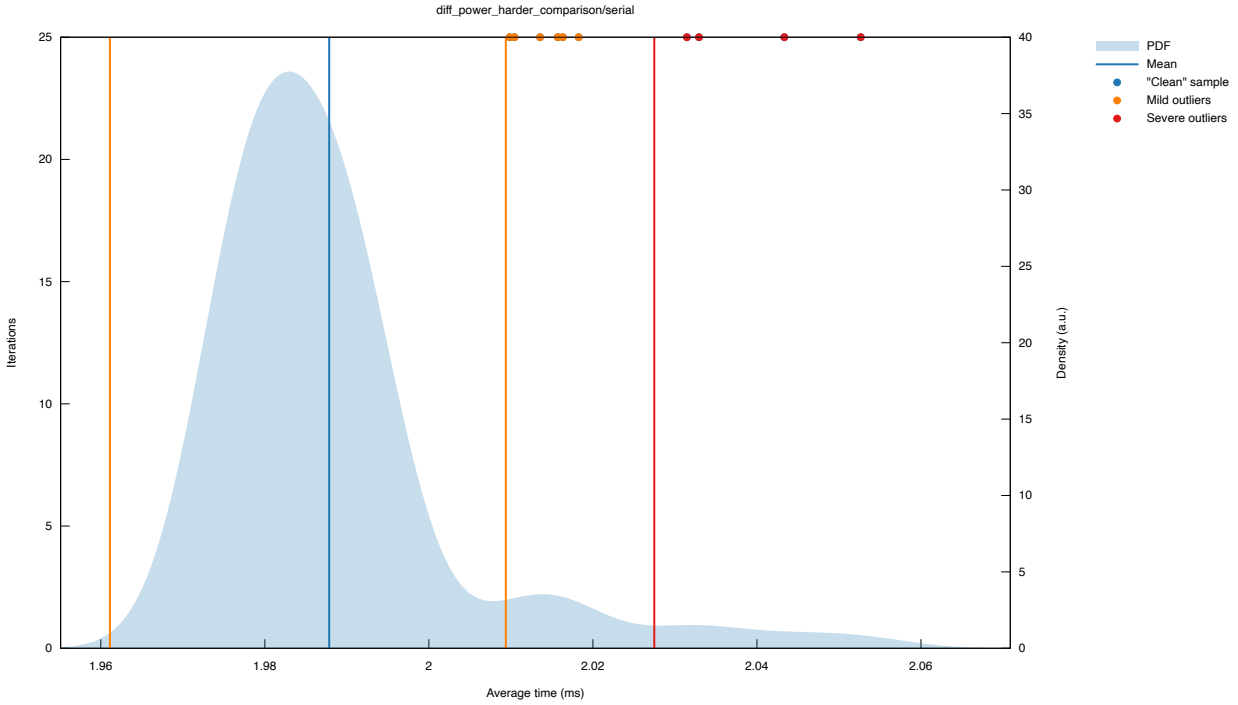


Figure 16: Runtime distribution of the serial execution of the `diff_power_harder` benchmark, generated by `criterion`. Note that this is correctly unchanged other than outliers from Figure 14, as it results from running the same test.

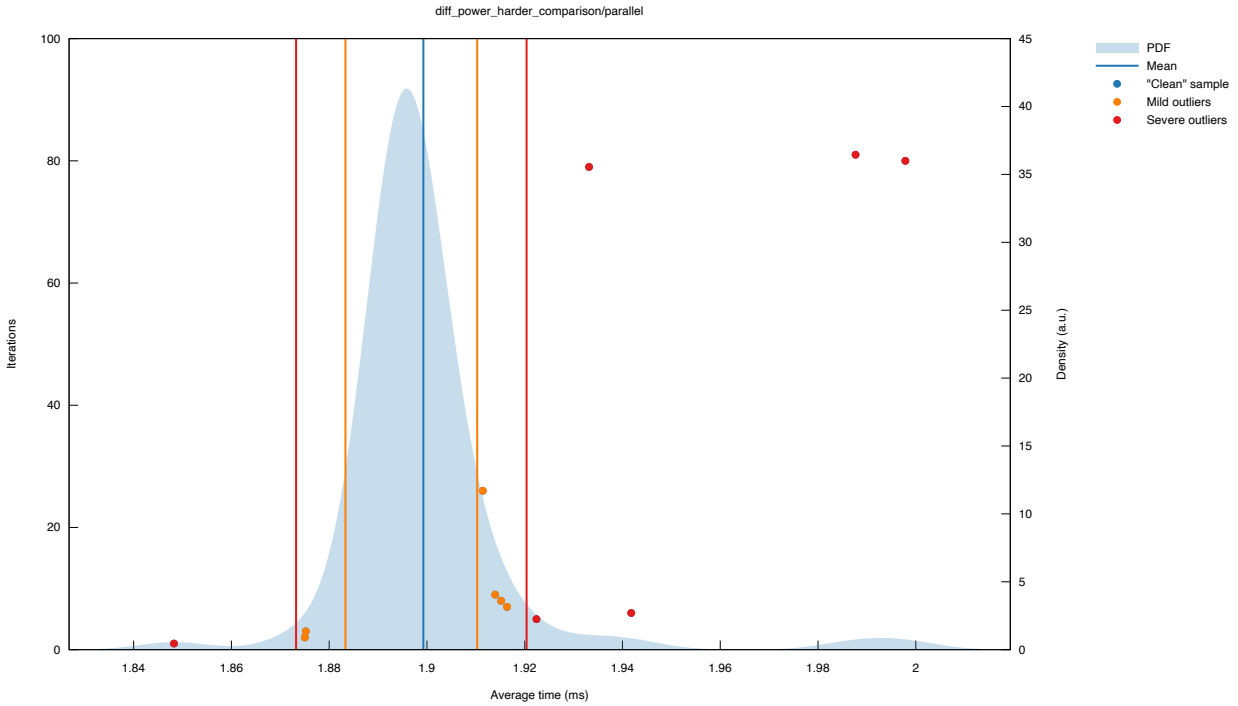


Figure 17: Runtime distribution of the parallel execution of the `diff_power_harder` benchmark, with the thread pool size constained to two, generated by `criterion`.

E Experiment hardware configuration

All experimental data was collected on an Apple M3 MacBook Air with 16GB of RAM. The output of the `lstopo` tool is shown in Figure 18 to facilitate experimental reproducibility.

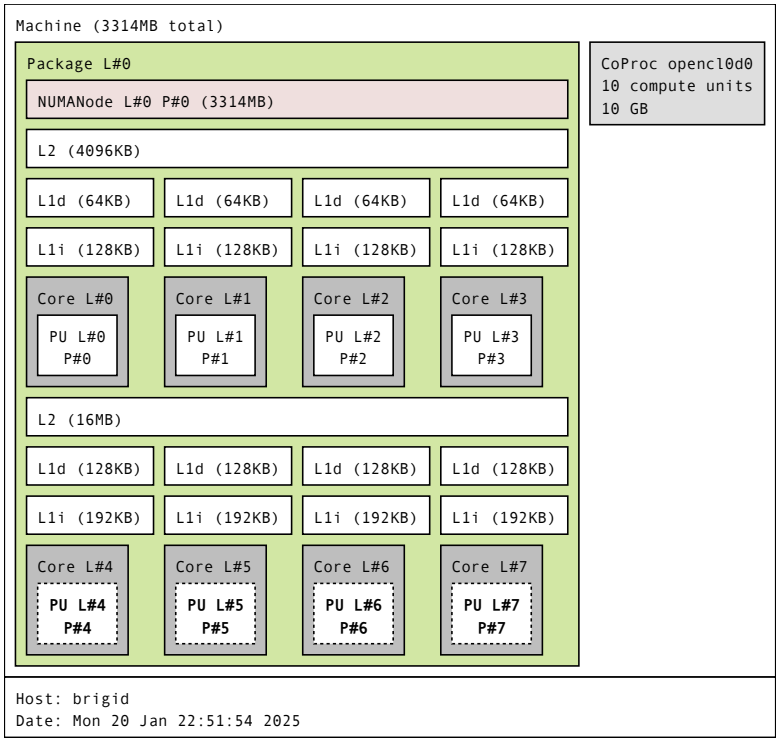


Figure 18: Output of the `lstopo` tool on the experimental machine.