ParEVO: Synthesizing Code for Irregular Data: High-Performance Parallelism through Agentic Evolution

Liu Yang [1] **Zeyu Nie** [1] **Andrew Liu** [1] **Felix Zou** [1] **Deniz Altinbüken** [2] **Amir Yazdanbakhsh** [2] **Quanquan C. Liu** [1]

Abstract

The transition from sequential to parallel computing is essential for modern high-performance applications but is hindered by the steep learning curve of concurrent programming. This challenge is magnified for **irregular data structures** (such as sparse graphs, unbalanced trees, and non-uniform meshes) where static scheduling fails and data dependencies are unpredictable. Current Large Language Models (LLMs) often fail catastrophically on these tasks, generating code plagued by subtle race conditions, deadlocks, and sub-optimal scaling.

We bridge this gap with **ParEVO**, a framework designed to synthesize high-performance parallel algorithms for irregular data. Our contributions include: (1) **The Parlay-Instruct Corpus**, a curated dataset of 13,820 tasks synthesized via a "Critic-Refine" pipeline that explicitly filters for empirically performant algorithms that effectively utilize Work-Span parallel primitives; (2) specialized **DeepSeek**, **Qwen**, and **Gemini** models fine-tuned to align probabilistic generation with the rigorous semantics of the ParlayLib parallel data structures and algorithms library; and (3) an **Evolutionary Coding Agent (ECA)** that significantly improves the "last mile" of correctness by iteratively repairing code using feedback from compilers, dynamic race detectors, and performance profilers.

On the ParEval benchmark, ParEVO achieves an average $106\times$ **speedup** (with a maximum of $1103\times$) across the comprehensive suite, and a robust $13.6\times$ **speedup** specifically on highly complex irregular graph problems, outperforming state-of-the-art commercial models like GPT-5-Thinking and Gemini-3-Pro. Furthermore, our evolutionary approach matches state-of-the-art expert *human-written* baselines, achieving up to a $4.1\times$ **speedup** on specific highly-irregular kernels (e.g., Maximal Independent Set). This demonstrates that AI-driven agents can effectively navigate the complex landscape of high-performance computing. Source code and datasets are available at `https://github.com/WildAlg/ParEVO` (ParEVO, 2026a).

[1]Department of Computer Science, Yale University, New Haven, CT, USA [2]Google DeepMind, Mountain View, CA, USA. Correspondence to: Quanquan C. Liu <quanquan.liu@yale.edu>.

Preprint. March 2, 2026.

1. Introduction

The breakdown of Dennard scaling and the subsequent stagnation of single-core frequency scaling have fundamentally shifted the computing paradigm. Performance improvements in modern software are now almost exclusively driven by parallelism, whether through multi-core CPUs, GPUs, or distributed clusters (Sahu et al., 2019). While "regular" parallelism (e.g., dense matrix multiplication) is well-understood and supported by mature libraries, **irregular parallelism** remains a grand challenge in High-Performance Computing (HPC).

Irregular algorithms, which operate on graph structures, sparse matrices, or adaptive meshes, are characterized by unpredictable memory access patterns and dynamic work distribution. In these regimes, the computational cost of processing a node or element depends on runtime data, making static load balancing ineffective. Writing efficient code for these problems requires sophisticated techniques like work-stealing, dynamic scheduling, and lock-free synchronization (Nichols et al., 2024).

Current Large Language Models (LLMs) struggle profoundly with this domain. Trained primarily on sequential Python or standard C++ code from GitHub, they exhibit strong "sequential bias." When asked to parallelize a graph traversal, they often attempt to wrap a standard Breadth-First Search (BFS) in a naive parallel loop (`#pragma omp parallel for`), ignoring the race conditions inherent in updating the 'visited' array. Alternatively, they may introduce coarse-grained locks that serialize execution, rendering the parallel code slower than its sequential counterpart (Kambhampati et al., 2024).

We argue that the solution lies not in teaching LLMs to write low-level threading primitives (like 'pthreads' or 'std::thread'), which are error-prone and hard to compose, but in leveraging high-level algorithmic primitives. **ParlayLib** (Blelloch et al., 2020) provides a suite of such primitives (e.g., 'filter', 'pack', 'scan', 'sort', 'reduce') that abstract away the complexities of scheduler management. By training LLMs to map natural language intent to these primitives, we can generate code that is *correct by construction* and mathematically provable to scale.

To this end, we introduce **ParEVO**, an end-to-end system for synthesizing high-performance parallel code. We detail the following main contributions:

- **Data-Centric Synthesis:** We introduce the *Parlay-Instruct* corpus, a dataset of 13,820 parallel coding tasks. Unlike previous datasets scraped from GitHub (which often contain broken code), our data is synthesized via a "Teacher-Student" pipeline and verified against a ground-truth compiler oracle. We provide a novel performance dataset generation technique focused on graph problems curated from the selection of well-known programming competitions curated by the online-judge DMOJ (DMOJ Developers, 2024).
- **DeepSeek-Parlay, Qwen-Parlay, Qwen-Rust, and Gemini-2.5-Parlay:** We release a fine-tuned 6.7B parameter Deepseek model for C++ (`https://huggingface.co/qqggez/deepseek-parlay-6.7b`) (ParEVO, 2026b), two fine-tuned 30B parameter Qwen3 models—one for C++ (`https://huggingface.co/qqggez/qwen3-30b-sft-stage2-merged`) (ParEVO, 2026) and one for Rust (`https://huggingface.co/YangLiuWillow/qwen3_rust_dpo_final_merged`) (ParEVO, 2026)—and a Gemini-2.5-Pro model fine-tuned for C++. These models outperform some larger closed-source and open-source models on parallel reasoning tasks by internalizing the data structures, semantics, primitives, and algorithms of the state-of-the-art ParlayLib library (Blelloch et al., 2020) and safe parallel Rust patterns.
- **Evolutionary Refinement:** We formalize both the data synthesis step and the final code generation process as an evolutionary search over the space of Abstract Syntax Trees (ASTs). Our agent generates a population of candidate solutions, compiles them, runs them against performance tests, and uses the error logs (or performance profiles) as "fitness functions" to drive mutation and crossover operations in the prompt space.
- **The Correctness-Speedup Trade-off:** We identify an "alignment tax" for concurrent programming. Our evaluation reveals that fine-tuning enables models to write significantly safer code (Pass@1 jumps from 0.42 to 0.76) at the expense of slightly slower peak performance (Speedup drops from $21.7\times$ to $13.6\times$). This trade-off occurs because fine-tuned models learn to conservatively avoid raw, risky atomics in favor of stable, high-level primitives (such as `parlay::unique`).

Specifically, our paper succeeds in the following task:

ParEVO democratizes parallel computing for irregular data by fine-tuning LLMs on verified primitives and deploying an evolutionary agent to iteratively optimize code based on runtime performance feedback.

All code can be found at `https://github.com/WildAlg/ParEVO` (ParEVO, 2026a).

2. Related Work

LLMs for Code Generation. Large Language Models have fundamentally shifted the landscape of software engineering, achieving remarkable success in sequential code completion (Husein et al., 2025), summarization (Ahmed & Devanbu, 2022), and translation (Eniser et al., 2024). Evaluation metrics have similarly evolved from surface n-gram overlap to structure-aware measures like CodeBLEU (Ren et al., 2020), which better correlate with functional correctness. However, current models struggle with complex planning and reasoning tasks (Kambhampati et al., 2024), a limitation that is magnified in High-Performance Computing (HPC). Nichols et al. (2024) demonstrated via the ParEval benchmark (Foundry et al., 2024) that while LLMs can generate syntactic structures for frameworks like Kokkos and MPI, they often fail to capture the semantic nuances of synchronization and race conditions. Recently, ParEval-Repo (Davis et al., 2025a;b) extended this evaluation to repository-level HPC translation tasks (e.g., multi-file codebases, build systems), highlighting that scaling beyond individual kernels introduces qualitatively different failure modes. Our work addresses this by moving beyond general-purpose pre-training, targeting the qualitatively harder regime of *parallel* and *irregular* algorithms where correctness requires respecting concurrency and performance depends on minimizing span.

Automated Parallelization and HPC Translation. Prior efforts in automated parallelization have largely focused on translating serial loops to OpenMP directives. BabelTower (Wen et al., 2022) previously tackled auto-parallelized program translation from sequential C to CUDA via a learning-based framework leveraging large-scale corpora and back-translation with reranking. OMPGPT (Chen et al., 2024) fine-tunes GPT-Neo to predict pragmas for regular loops, while AutoParLLM (Mahmud et al., 2023) uses Graph Neu-

ral Networks to guide LLM generation based on parallelism patterns. TehraniJamsaz et al. (2024) attempts unsupervised translation between languages and their HPC extensions (CodeRosetta, 2024) but lacks a feedback mechanism for correctness. To address correctness risks such as subtle parallel bugs that often arise in serial-to-CUDA/OpenMP translation, MuSL (Ke et al., 2025; Ke, 2025; kcxain, 2025) proposed a mutual-supervision loop where a translator and a test-generator co-evolve: the tester synthesizes unit tests to filter translations, and the translator produces code to improve the tester. More recently, UniPar (Bitan et al., 2025) introduced a multi-agent framework for translating code between serial, OpenMP, and CUDA formats. While UniPar evaluates functional correctness (achieving 33%), it does not explicitly optimize for or benchmark the runtime scalability (work-span) of the generated algorithms. In contrast, ParEVO specifically targets *irregular* data, such as graph traversals and sparse matrix operations, where correct translation is insufficient, and *performance speedup* via parallelism, software engineering techniques, and performant algorithms and data structures is key. Recent advances have further specialized LLMs for parallel domains. For instance, Chaturvedi (2024) successfully fine-tuned base models on the HPC-Instruct dataset to target low-resource parallel languages, demonstrating that smaller, specialized models can match proprietary models on the ParEval benchmark. Similarly, frameworks like MARCO (Rahman, 2025) and PerfCoder (Yang, 2025) utilize multi-agent reasoning and execution trajectories to separate code generation from performance tuning. However, while these frameworks primarily target traditional imperative paradigms like OpenMP and CUDA, ParEVO specifically targets the algorithmic complexities of irregular data by grounding the model in the composable semantics of ParlayLib.

Structured Reasoning and Agentic Coding. To transcend the stochastic limitations of single-shot generation, frameworks like Reflexion (Shinn et al., 2023) use verbal reinforcement to iteratively correct failures. More recently, this paradigm has been extended via evolutionary search. Building upon this, EvoTune (Surina et al., 2025) augments LLM-based evolutionary program search by periodically updating the model via reinforcement learning on search-derived signals. Similarly, AI tree search systems (Aygün et al., 2025) embed LLM-based code mutation within a search procedure to maximize a measurable quality metric. To benchmark these search processes, AlgoTune (Press et al., 2025a;b) introduced a suite for numerical programs and evaluated an agent that iterates by editing, compiling, timing, and selecting the fastest valid variant. Concurrent open-source works such as OpenEvolve (Sharma, 2025a) have demonstrated the efficacy of coupling LLMs with genetic algorithms (Assumpção et al., 2025; Khrulkov et al., 2025; Novikov et al., 2025) and Quality-Diversity metrics (e.g., MAP-Elites) to prevent diversity collapse during program synthesis. ParEVO brings this evolutionary paradigm to the HPC domain, replacing standard unit-test fitness functions with rigorous hardware profiling and sanitizer-based data race detection.

Abstractions for Irregular Parallelism. A core theme in parallel algorithmics is that abstraction choice determines accessibility. The classic work-span model (Brent, 1974) and work-stealing schedulers (Blumofe & Leiserson, 1999) provide a principled foundation for nested parallelism. High-level libraries like ParlayLib (Blelloch et al., 2020; Anderson et al., 2020) expose this theory through composable primitives (e.g., `scan`, `reduce`, `filter`), making provably efficient algorithms more accessible. Similarly, specialized abstractions such as GraphIt (Zhang et al., 2018; GraphIt-DSL et al., 2018) separate algorithm specification from scheduling choices to enable systematic performance tuning for irregular graph workloads, while Ligra (Shun & Blelloch, 2013) provides a lightweight shared-memory graph processing framework with simple vertex/edge mapping primitives and density-adaptive traversal strategies. Benchmarks like PBBS (Shun et al., 2012; Anderson et al., 2022) and Rusty-PBBS (Abdi et al., 2023a) formalize the evaluation of these irregular workloads. ParEVO leverages these insights by training models to target primitive-based code-writing within the Parlay ecosystem, ensuring that generated code is not just a parallel loop, but a structurally sound parallel algorithm capable of handling load imbalance inherent in irregular data (Sahu et al., 2019; Bronson et al., 2013).

Test-Time Compute and Execution Feedback. A growing consensus indicates that standard Supervised Fine-Tuning (SFT) and text-based reflection are insufficient for generating highly optimized code. Consequently, the field has rapidly shifted toward integrating real-machine execution feedback into the LLM reasoning loop. Using empirical hardware profiling as a direct reward signal drastically improves kernel efficiency (Du et al., 2025; Merouani et al., 2025; Lei et al., 2025). Crucially, Singh et al. (2024) applied test-time program search to the ParEval benchmark and empirically proved that LLMs exhibit a severe capability gap when attempting to act as their own "verifiers" for parallel code. This limitation directly motivates ParEVO's Evolutionary Coding Agent (ECA), which sidesteps the unreliable "LLM-as-a-judge" paradigm in favor of treating deterministic compilers and sanitizers as ground-truth adversarial critics.

3. Methodology: The ParEVO System

ParEVO is composed of three distinct stages: (1) Data Synthesis through Evolutionary Search, (2) Supervised Fine-Tuning, and (3) Inference-Time Evolutionary Search.

3.1. Stage 1: The Parlay-Instruct Fine-Tuning Dataset Corpus

The primary bottleneck for training "HPC-aware" LLMs is data scarcity. High-quality parallel C++ code is rare on GitHub compared to React components or Python scripts. We generated a synthetic dataset which incorporates parallel performance constructs, syntax, software engineering techniques, data structures, and algorithms, using a "Teacher-Student-Critic" pipeline via OpenEvolve (Sharma, 2025b). This synthetic dataset contains three parts: (1) the ParlayLib primitives, (2) DMOJ slow-fast code comparison pairs, and (3) DMOJ problem-solution pairs with labeled status, runtime performance, and any compiler or runtime error messages.

3.1.1. Seed Generation and Mutation

We manually authored 593 "golden" examples covering ParlayLib's core primitives and 20 problems from DMOJ (DMOJ Developers, 2024). We then used Gemini-3-Pro (the "Teacher") to mutate these seeds. We defined three mutation operators $\mathcal{M}$:

1. **Type Mutation ($\mathcal{M}_{type}$):** Changes the underlying data type (e.g., 'int' $\rightarrow$ 'std::string' or custom 'struct Point'). This forces the model to learn C++ template instantiation rules.
2. **Constraint Mutation ($\mathcal{M}_{cons}$):** Adds logical predicates (e.g., "Sort only odd numbers" $\rightarrow$ requiring a 'filter' then 'sort'). This forces the composition of primitives.
3. **Algorithmic Mutation ($\mathcal{M}_{algo}$):** Transforms the problem structure, e.g., converting a 'reduce' problem into a 'scan' (prefix sum) problem.

3.1.2. The Critic Loop: Rejection Sampling

Let P be a generated problem and C be the generated code. We accept (P, C) into the dataset if and only if:

$$\text{Compile}(C) \wedge \text{UnitTest}(C) \tag{1}$$

That is, we only accept code that compiles and passes the unit tests.

Data Verification Pipeline. To ensure the integrity of the *Parlay-Instruct* corpus, we employed a strict *execution-based verification protocol*. Unlike prior datasets that rely solely on static analysis, every generated candidate in our pipeline was compiled against the ParlayLib headers and executed against a synthesized unit test. Candidates that failed compilation or runtime assertion checks were automatically discarded. This rigorous filtration process yielded 13,820 verified instruction-tuning pairs, which we partitioned into a fine-tuning training set of size 13,120 and a held-out test set of 700 pairs for evaluation.

Example Data Point from *Parlay-Instruct*

Instruction:
Filter even numbers from 0 to 100,000 using `parlay::delayed::filter_op` and materialize the result.

Target Code (Verified):

```
auto seq = parlay::iota(100000);
// specific delayed filter operation
auto f = parlay::delayed::filter_op(
    seq, [](int x) {
    return x%2==0 ? std::optional{x}
    : std::nullopt;
});
auto s = parlay::delayed::to_sequence
    (f);
```

Hidden Unit Test:

```
1 int main() {
2     // ... [setup code injected here] ...
3     assert(s.size() == 50000);
4     for(int i=0; i<50000; i++) assert(s[i] == i
      *2);
5     std::cout << "Passed";
6 }
```

Figure 1. A representative sample from the training corpus. Each sample includes a natural language instruction, the ground-truth parallel implementation, and an executable unit test used for verification.

Performance Optimization Dataset. To enable the model to reason about runtime efficiency, we curated a benchmark of 20 challenging graph problems from the DMOJ competitive programming platform (DMOJ Developers, 2024). We synthesized optimization trajectories for these problems using the OpenEvolve framework (Sharma, 2025b) powered by Gemini-3-Pro. The data generation process followed the following novel protocol:

1. **Agent Initialization:** The agent was provided with the problem description and ParlayLib documentation, with a dual objective function minimizing both test failures and execution time.
2. **Trajectory Extraction:** We recorded the agent's iterative refinements, extracting pairs of solutions $(C_{\text{base}}, C_{\text{opt}})$ from the evolutionary history.
3. **Speedup Threshold:** To ensure high-quality training signal, we filtered for pairs where the optimized solution C_{opt} achieved a runtime speedup of at least $1.2\times$ over C_{base}.

We constructed pairwise comparison examples using the solution pairs identified in the previous step. To eliminate

positional bias, we randomized the assignment of "Code A" and "Code B" so that the faster implementation appears in either position with equal probability. The model is trained to identify the more performant solution using the following format:

> **Instruction:** Determine which of the two code solutions has better performance.
>
> **Input:**
> Code A: [Source Code]
> Code B: [Source Code]
>
> **Output:** [Label of the Faster Solution]

A concrete example of this comparison format is provided in Figure 2.

While learning on performance edits has been used in (Shypula et al., 2024), our dataset is distinct in its focus on the complex, global transformations required for *irregular parallelism*, rather than the local sequential optimizations primarily targeted in the prior work.

Rust Parlay Primitives Given that the distinct Rust primitives were insufficient to constitute a robust fine-tuning dataset, we opted to include them directly in the context window. This approach allowed us to leverage the models' pattern-matching and in-context learning capabilities without the need for parameter updates. To support this process, we integrated a full suite of Parlay-equivalent Rust primitives derived from **RPB** (Abdi et al., 2023b). Furthermore, to support the generation of higher-complexity algorithms, we manually implemented the delayed execution primitives in Rust and supplied them as immutable reference implementations within the system prompt.

Rust Evolutionary Dataset. To train the evolutionary coding agent for the Rust domain, we constructed a specialized dataset derived from the DMOJ benchmark execution logs. We aggregated the raw logs to extract code solutions, runtime metrics, and error traces. The data underwent a rigorous cleaning pipeline: we first filtered out irrelevant infrastructure failures (e.g., permission errors) and removed the held-out test set. We then deduplicated the remaining entries, prioritizing successful submissions while retaining a diverse set of failing attempts characterized by distinct error messages. The final corpus was serialized into JSONL format, where each entry explicitly pairs a problem description with the corresponding code, execution status, runtime performance, and any resulting compiler or runtime error messages. This rich metadata distinguishes our dataset from standard code corpora, enabling the model to learn both correct optimization patterns and specific error-correction strategies. Such a detailed corpus of training data is necessary for Rust given that Rust is notoriously difficult to use for irregular parallelism; hence, the available training data (including errors and compile-time messages) is rare for this language in the available base models.

3.2. Stage 2: Fine-Tuning DeepSeek, Gemini-2.5, Qwen3 for ParlayLib and Rust RPB

We selected **DeepSeek-6.7b-base** and **Qwen3-Coder-30B-A3B-Instruct** as our open-source backbones due to their strong performance on standard C++. These models represent a tiered architecture strategy: DeepSeek-6.7b serves as our efficient, lightweight baseline, while Qwen3 acts as our high-capacity large model. We fine-tuned the model using Low-Rank Adaptation (LoRA) (Hu et al., 2022) to minimize compute costs while preserving the base model's reasoning capabilities. We selected **Gemini-2.5-Pro** as our third base model due to its extensive context window for handling complex, long-context scenarios. All three models underwent fine-tuning to align them with our specific domain requirements.

Training Configuration. We configured the training pipeline according to model scale. For **DeepSeek-6.7b-base**, we executed single-stage Supervised Fine-Tuning (SFT) on an NVIDIA RTX 5000 Ada machine. We targeted the query and value projections using LoRA ($r = 8, \alpha = 16$) and trained on a combined dataset of ParlayLib syntax and 'slow-fast' performance pairs (FP16, learning rate 2e-4).

For the larger **Qwen3-Coder-30B-A3B**, we implemented a dual-stage alignment pipeline on an NVIDIA H200 GPU. The first stage established domain capability via SFT on ParlayLib syntax and standard DMOJ solutions, using QLoRA ($r = 16, \alpha = 32$) across all linear attention and MLP layers. The second stage applied Direct Preference Optimization (DPO) to explicitly suppress failure modes. In this phase, we trained on contrastive triplets (pairing passing solutions against failing or inefficient implementations) using a reduced learning rate of 5e-6 and $\beta = 0.1$.

Evaluation Environment. Performance benchmarks were conducted on a dual-socket compute node featuring two Intel Xeon Platinum 8562Y+ processors (64 physical cores total). To ensure consistent comparisons across frameworks, all experiments use 32 threads for OpenMP, ParlayLib, and Rust unless otherwise specified.

3.3. Stage 3: Evolutionary Coding Agent (ECA)

Evolutionary Search Strategy. To transcend the stochastic limitations of single-shot generation, we deploy an evolutionary agent that iteratively refines code for both correctness and performance. We model this process as a directed population-based search in the discrete space of possible programs. See Figure 3 for a diagram of the workflow.

Code A (Efficient): Parallel Map-Scan-Write

```cpp
// 1. MAP: Count events in parallel (No Locking)
parlay::sequence<int> counts(N);
parlay::parallel_for(0, N, [&](int r) {
    int cnt = 0; /* logic checks ... */
    if (valid) cnt++;
    counts[r] = cnt;
});

// 2. SCAN: Calculate offsets (Prefix Sum)
auto [offsets, total] = parlay::scan(counts);

// 3. WRITE: Parallel Fill (Zero Realloc/ Contention)
parlay::sequence<Event> evs(total); // Alloc exact size
parlay::parallel_for(0, N, [&](int r) {
    int k = offsets[r];
    /* logic checks ... */
    if (valid) evs[k++] = {u, v, t};
});

```

Code B (Inefficient): Sequential Push

```cpp
// 1. Setup Vector (Heuristic reservation)
std::vector<Event> events;
events.reserve(2 * N * N); // May still realloc

// 2. Iterate Sequentially (Cannot Parallelize)
for (int r = 0; r < N; ++r) {
    for (int c = 0; c < N; ++c) {
        /* logic checks ... */

        // BOTTLENECK: Single thread, capacity checks,
        // and reallocation overhead.
        if (valid) {
            events.push_back({u, v, t});
        }
    }
}

```

Figure 2. **Comparison of Event Generation Strategies. Left:** Code A employs a Map-Scan-Write pattern to enable lock-free parallel writing. **Right:** Code B relies on sequential `push_back`, preventing parallelization and incurring reallocation costs.

The agent maintains a diverse population of candidate solutions, each associated with specific performance metrics (test coverage, execution time) and diagnostic artifacts (compiler logs, failure reasons, and targeted refinement instructions). The search initializes with either a baseline functional solution or a raw problem description. We define the fitness function $f(x)$ for a candidate solution x as:

$$f(x) = \begin{cases} 0 & \text{if } x \text{ fails compilation or tests} \\ \frac{1}{T(x)+\varepsilon} & \text{if } x \text{ passes, where } T(x) \text{ is runtime} \end{cases} \tag{2}$$

Furthermore, candidate solutions that trigger data races or deadlocks caught by dynamic analysis are assigned a fitness of 0.

A critical design choice in our evolutionary loop is the reliance on deterministic external tools—specifically compilers and dynamic race detectors—rather than LLM-based static analysis. Because LLMs process code as a sequence of text tokens, they natively fail to capture inter-thread timing and synchronization structures, making them highly susceptible to hallucinating data races. Furthermore, as noted by Singh et al. (2024), LLMs are fundamentally unreliable verifiers of low-level parallel code. By utilizing dynamic race detection as an absolute, non-negotiable filter in our fitness evaluation, we guarantee that the LLM is forcibly corrected whenever it hallucinates unsafe memory accesses.

In each generation, the agent selects survivors to populate the context window for the next iteration. To balance exploitation and exploration, we select the top $k = 3$ solutions by fitness (performance) and $d = 5$ diverse solutions via MAP-Elites. The MAP-Elites algorithm maintains diversity by categorizing solutions into an archive based on predefined feature dimensions; in our implementation, we characterize solutions by their code length, cyclomatic complexity, and the frequency of synchronization primitives (e.g., locks vs. atomic operations). These selected candidates, along with their diagnostic artifacts, prompt the LLM to synthesize the next generation of improved code. The process terminates by returning the candidate with the maximum fitness score.

3.4. Supported Languages

To demonstrate the versatility of our approach, in this paper, we use our ParEVO on two languages: C++ and Rust. For C++, we use our ParEVO system to fine-tune models on ParlayLib (Blelloch et al., 2020). For Rust, we use our ParEVO system to fine-tune models on RPB: Rust Parallel Benchmarks Suite (Abdi et al., 2023a;b). For both C++ and Rust, our methods lead to improved performance.

3.5. Benchmarking Suite

We evaluate our framework across four distinct benchmarks to assess both generation quality and runtime performance. First, we compare our fine-tuned models against state-of-the-art local and commercial LLMs using the **ParEval** (Nichols et al., 2024) library. Second, we measure absolute perfor-

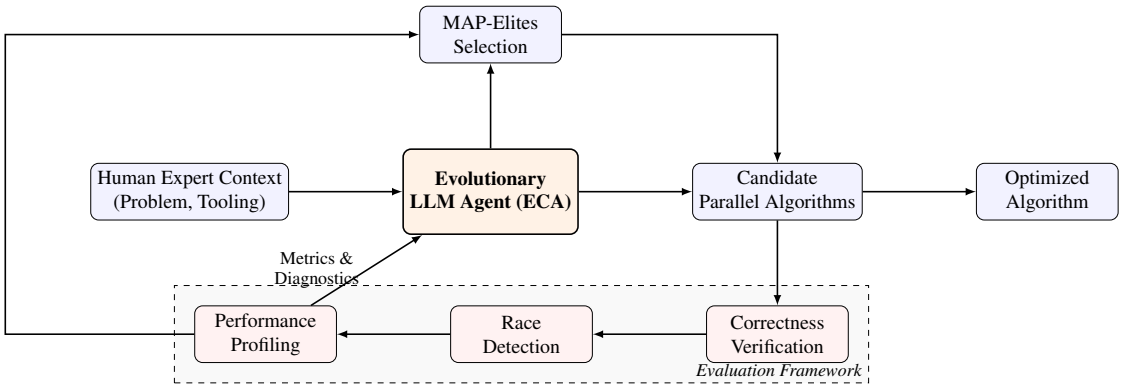


Figure 3. **Overview of the ParEVO Framework.** The system integrates human expert context (problem formulation, parallel tooling) with an evolutionary LLM agent. The cycle iteratively refines candidate parallel algorithms through a rigorous evaluation framework (correctness verification, dynamic race detection, and performance profiling), using metrics to guide the selection of the next population via MAP-Elites.

mance against expert human baselines, utilizing C++ solutions from **PBBSBench** (Shun et al., 2012) and Rust implementations from **RPB** (Abdi et al., 2023b). Finally, to test generalization, we evaluate on a held-out set of **DMOJ** competitive programming problems. In this setting, we compare the runtime of code generated by ParEVO against official contest solutions, demonstrating significant speedups.

4. Experimental Results

4.1. Experimental Setup

Hardware. All experiments were conducted on a dual-socket compute node equipped with two Intel Xeon Platinum 8562Y+ processors (64 physical cores total) and 512GB DDR5 ECC RAM. An NVIDIA H200 GPU was utilized solely for inference.

Benchmarks. We evaluated on:

1. **ParEval:** The ParEval testing suite of (Nichols et al., 2024).
2. **PBBSBench & RPB:** Expert-written C++ and Rust baselines (Shun et al., 2012; Abdi et al., 2023b).
3. **DMOJ:** A held-out set of competitive programming problems. (DMOJ Developers, 2024)

4.2. Main Results: ParEval Performance

Methodological Note on Expected Speedup. In traditional systems literature, the geometric mean is typically used to average normalized execution times of a static benchmark suite across different hardware. However, in the context of zero-shot code generation over a large distribution of tasks (ParEval), we conceptualize performance formally as an expected capability reward. Specifically, we report the arithmetic mean of `Speedup@1` to represent the *expected speedup* ($\mathbb{E}[S]$) a user would experience when querying the model with a random task from the problem domain. This aligns directly with standard machine learning evaluation practices for reporting expected test-time rewards over a distribution, as opposed to summarizing the total execution time of a fixed static workload.

Table 1 presents the performance of local and commercial models. Our fine-tuned models (**Gemini-2.5-Parlay** and **DeepSeek-Parlay**) significantly outperform their base counterparts. Notably, **Gemini-2.5-Parlay** achieves an average $106\times$ speedup over the baseline, driven by its ability to generate valid, compilable parallel code (`Build@1` 0.84 vs 0.25 of the state-of-the-art `Gemini 3.0 Pro`). Even our smallest fine-tuned model, DeepSeek-Parlay (with 6.7b parameters) is able to beat the commercial state-of-the-art `Gemini-3-Pro`.

Execution Model	Code	Sched.	Build@1	Pass@1	Speedup
Claude Opus 4.5	Parlay	Parlay	0.28	0.27	0.65
GPT-5 Thinking	Parlay	Parlay	0.73	0.63	14.03
Gemini-2.5-Flash	Parlay	Parlay	0.58	0.29	13.42
Gemini-2.5-Pro	Parlay	Parlay	0.98	0.77	10.40
Gemini-3-Pro	Parlay	Parlay	0.25	0.23	12.29
Gemini-2.5-Parlay	**Parlay**	**Parlay**	**0.84**	**0.33**	**106.87**
DeepSeek-6.7B-Base	Parlay	Parlay	0.89	0.11	3.65
DeepSeek-Syntax	Parlay	Parlay	0.85	0.12	6.60
DeepSeek-Parlay	**Parlay**	**Parlay**	**0.79**	**0.35**	**16.40**
Qwen3-Parlay	**Parlay**	**Parlay**	**0.50**	**0.33**	**8.63**
DeepSeek-Coder-V2-Lite-Base	Parlay	Parlay	0.80	0.09	2.57
Qwen2.5-Coder-32B	Parlay	Parlay	0.93	0.11	9.98
Qwen2.5-Coder-32B-Instruct	Parlay	Parlay	0.61	0.41	12.91
DeepSeek-Coder-V2-Lite-Base	Rust	Rayon	0.73	0.29	6.26
DeepSeek-Coder-V2-Lite-Instruct	Rust	Rayon	0.40	0.02	0.77
Qwen2.5-Coder-32B	Rust	Rayon	0.82	0.45	5.64
Qwen2.5-Coder-32B-Instruct	Rust	Rayon	0.63	0.49	5.97
Qwen3-Coder-30B-Instruct	Rust	Rayon	0.61	0.50	5.70
Qwen3-Rust	**Rust**	**Rayon**	**0.64**	**0.46**	**6.10**
StarCoder2-15B	Rust	Rayon	0.77	0.27	3.58
Gemini-3-Pro	Rust	Rayon	0.97	0.82	7.42

Table 1. ParEval results (Temperature=0.2). Our fine-tuned models demonstrate orders-of-magnitude improvements in speedup.

Impact of Fine-tuning on Code Quality. As illustrated in Figure 4, fine-tuning yields a dramatic improvement across all three performance metrics. The base Gemini-2.5-Pro model frequently struggles with the strict type system of parallel libraries, resulting in low compilation rates (`Build@1`). In contrast, ParEVO demonstrates a near-perfect `Build@1` rate, indicating that the model has successfully internalized the syntactic constraints of ParlayLib. This syntactic grounding translates directly into algorithmic efficacy: the fine-tuned model not only generates compilable code but consistently selects efficient parallel patterns, driving substantial gains in both functional correctness (`Pass@1`) and runtime performance (`Speedup@1`), where it achieves orders-of-magnitude improvements over the baseline.

4.3. Semantic Alignment via Fine-Tuning

A critical advantage of ParEVO is its ability to learn the correct semantics of parallel primitives. In the complex number sorting task (Figure 5), the base model failed completely (Build@1 = 0), struggling with C++ custom comparators. The fine-tuned model not only compiled (Build@1 = 1) but achieved a speedup of 17.5×. This suggests that the model has learned to navigate the complex type system of ParlayLib.

4.4. Performance Analysis: Strong Scaling

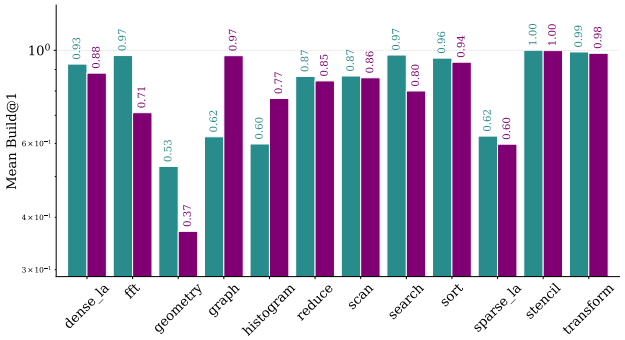
Code correctness is insufficient for HPC; the solution must also scale. Figure 6 demonstrates strong scaling up to 64 cores. For regular parallelism like Discrete Fourier Transform, our model generates code that scales near-linearly (40× speedup), abstracting away complex synchronization that typically hinders manual implementations.

4.5. Comparison vs. Expert Baselines

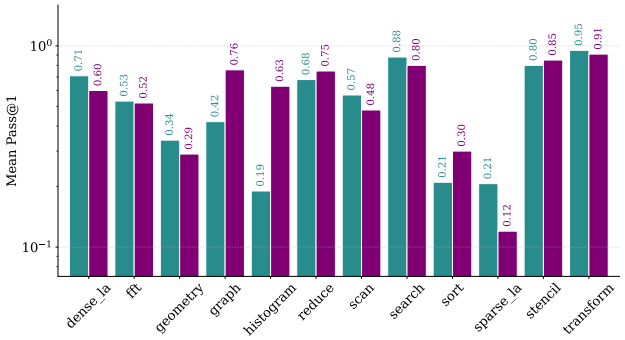
We benchmarked our generated solutions against expert human implementations from PBBSBench (C++) and RPB (Rust). As shown in Table 2, ParEVO matches or exceeds expert performance. For **Maximal Independent Set**, the generated Rust solution achieved a 4.1× speedup over the baseline by identifying a superior parallel strategy. We demonstrate the maximum speedups we can gain by using `Gemini-3-Pro` with our ParEVO evolutionary strategy described in Section 3.3.

4.6. Ablation Study: Evolutionary Agent

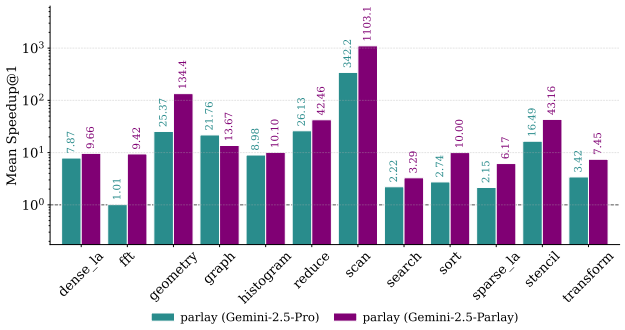
To isolate the contribution of the Evolutionary Coding Agent (ECA), we evaluated performance with the agent disabled. Table 3 confirms that iterative refinement is crucial: 30 iterations of ECA yield a 2.2× performance multiplier over single-shot generation.



(a) Mean Build@1



(b) Mean Pass@1



(c) Mean Speedup@1

Figure 4. ParEval Metrics Comparison between Gemini-2.5-Pro and Gemini-2.5-Parlay. (a-c) highlight that fine-tuning significantly improves the model's ability to construct valid ParlayLib code, with substantial gains in build and pass rates as well as improved running time over the base model.

For this ablation study, we utilized a reserved set of training problems sourced from DMOJ[1]. In Table 3, the base performance (1.00×) corresponds to `baseline.cpp`, which we define as the very first solution that passes all tests (this is not necessarily the solution from iteration 1 if the early attempts fail). The reported speedup for the ECA configurations is calculated as the average of the relative speedups achieved across all datasets within this training corpus.

[1]The DMOJ training problems are available at https://github.com/WildAlg/ParEVO/tree/main/code-contests-dataset

Input Prompt:

```
#include <parlay/primitives.h>
#include <complex>
/* Sort vector x of complex numbers by
    magnitude (ascending) using ParlayLib.
    */
void sortComplexByMagnitude(parlay::
    sequence<std::complex<double>>& x) {
```

Base Model (Gemini-2.5-Pro)

```
{
    // Build Error: parlay::sort expects
    // different comparator signature
    parlay::sort(x, [](const std::complex<
    double>& a, const std::complex<double>&
     b) {
        return std::abs(a) < std::abs(b);
    });
}

```

Status: Compilation Failed (Build@1=0)
Fine-tuned Model (ParEVO)

```
{
    // Correct: uses sort_inplace and
    // generic auto params for complex type
    parlay::sort_inplace(x, [](const auto&
    a, const auto& b) {
        return std::abs(a) < std::abs(b);
    });
}

```

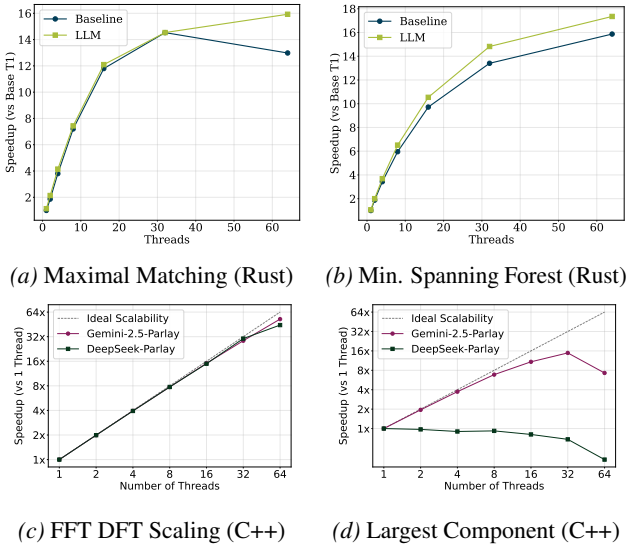
Status: Passed (17.5× *Speedup*)

Figure 5. **Semantic Alignment Example.** The base model (top) fails to compile due to incorrect API usage and strict type definitions in the lambda. The fine-tuned model (bottom) correctly identifies `sort_inplace` and uses `auto` to handle the complex number types safely.

While the actual prompt in each iteration contains more context (other iterations/metrics) by the default template of `openevolve`, the structural system prompt we specified for the ECA is provided in Appendix A.

4.7. Analysis: The Correctness-Speedup Trade-off

A deeper analysis of Graph problems (Table 4) reveals a trade-off. Fine-tuning increases correctness (`Pass@1` 0.42 → 0.76) by enforcing safe API usage, but this sometimes degrades peak speedup (21× → 13×) as the model favors stable, high-level primitives (e.g., `parlay::unique`) over risky, fine-grained atomic operations.



(a) Maximal Matching (Rust) *(b)* Min. Spanning Forest (Rust)

(c) FFT DFT Scaling (C++) *(d)* Largest Component (C++)

Figure 6. Strong Scaling results. (c) Algorithms like Discrete Fourier Transform show excellent scaling with ParEVO's generated code, reaching nearly 40× speedup on 64 cores.

Table 2. Runtime Comparison: PBBS & RPB at Thread=32, best speedup across test inputs. Baseline code is state-of-the-art human-written code. We also demonstrate the speedup against one thread, labeled Speedup (1T).

Problem	Model/Method	Language	Runtime (s)	Speedup (1T)	Speedup (Base)
Maximal Independent Set	Baseline	Rust	0.31876	1.116×	–
Maximal Independent Set	PAREVO (GEMINI)	Rust	0.07728	0.938×	4.125×
Maximal Matching	Baseline	Rust	0.20646	21.723×	–
Maximal Matching	PAREVO (GEMINI)	Rust	0.1928	21.43286835×	1.0708×
Minimum Spanning Forest	Baseline	Rust	0.41968	13.427×	–
Minimum Spanning Forest	PAREVO (GEMINI)	Rust	0.38004	13.87×	1.1043×
Spanning Forest	Baseline	Rust	0.11571	10.378×	–
Spanning Forest	PAREVO (GEMINI)	Rust	0.08865	15.482×	1.3052×
Minimum Spanning Forest	Baseline	C++	1.24	22.633×	–
Minimum Spanning Forest	PAREVO (GEMINI)	C++	1.169	23.689×	1.061×
Histogram	Baseline	C++	0.051	27.59×	–
Histogram	PAREVO (GEMINI)	C++	0.019	> 13.94×	2.68421×
Plane Sweep	Baseline	C++	131.695	27.59×	–
Plane Sweep	PAREVO (GEMINI)	C++	6.627	> 1.814×	2.68421×

5. Discussion and Limitations

5.1. The Role of Abstraction in Parallelization

Our findings suggest that the efficacy of LLM parallel code generation is heavily contingent on the level of abstraction provided by the target intermediate representation (IR). We argue that the superior performance of ParEVO on ParlayLib stems from an *alignment of abstraction*. Imperative models like OpenMP force the LLM to manage global state and explicit synchronization: tasks that maximize the "state-tracking" burden on the attention mechanism and increase the probability of race conditions.

In contrast, ParlayLib functions as a high-level parallel DSL. Its functional primitives (e.g., `map`, `reduce`, `scan`) encapsulate complex scheduling logic and enforce immutability. This reduces the problem of parallelization to *local* transformations (mapping serial loops to equivalent functional constructs) which aligns naturally with the token-local prediction capabilities of Transformer models.

Table 3. Impact of Evolutionary Refinement (ECA) on Speedup. 30 iterations of ECA yield a 2.2× speedup over the first valid solution.

CONFIGURATION	SPEEDUP
GEMINI-3-PRO (NO ECA)	1.00× (BASELINE)
GEMINI-3-PRO + ECA (10 ITER)	1.498×
GEMINI-3-PRO + ECA (30 ITER)	**2.218×**

Table 4. Performance on Graph Problems. Fine-tuning improves reliability (Pass@1) but favors safer, slightly slower algorithms.

Model	Build@1	Pass@1	Speedup@1
Gemini-2.5-Pro	0.62	0.42	**21.76**
Gemini-2.5-Parlay	**0.97**	**0.76**	13.67

By training our models to target ParlayLib's composable primitives, ParEVO naturally aligns the optimization objective with the token-local reasoning capabilities of the Transformer architecture, yielding code that is both mathematically sound and highly performant.

5.2. Limitations and Future Directions

- **Architectural Scope:** ParEVO is currently optimized for shared-memory multicore architectures. It does not address the distributed memory paradigm (e.g., MPI/PGAS), where communication latency and data partitioning introduce a distinct set of optimization constraints.
- **Inference Latency vs. Runtime Efficiency:** The Evolutionary Coding Agent trades inference-time compute for execution-time speedup. While the cost of generating multiple candidates and compiling them is nontrivial, we argue this is an acceptable amortized cost for HPC kernels that may run trillions of times over their lifecycle.
- **Domain Generalization:** As observed in some benchmarks, the model can suffer from "confident hallucinations" when applying learned parallel patterns to unfamiliar algorithmic domains. Future work will investigate integrating formal verification tools into the evolutionary loop to constrain these semantic errors.

6. Conclusion

We have presented **ParEVO**, a framework that bridges the gap between modern generative AI and high-performance computing. By curating a specialized dataset of parallel primitives and fine-tuning models to internalize the **Work-Depth** cost model, we achieve state-of-the-art results on the ParEval benchmark, surpassing both commercial LLMs and traditional heuristics.

Crucially, our results demonstrate that syntax generation alone is insufficient for HPC. The integration of an **Evolutionary Coding Agent**—which treats the compiler and runtime profiler as adversarial critics—is essential for traversing the optimization landscape. This work creates a precedent for **AI-Driven Performance Engineering**: moving beyond simple code completion to systems that actively reason about scalability, correctness, and the complex interplay between algorithms and hardware.

Acknowledgements

We thank Lin Zhong for helpful discussions and Rust resources, Ramla Ijaz for helpful discussions, and Roger Fu for compiling and providing to us the publicly available test cases for the competitive programming problems we used.

We also thank the extended team at Google DeepMind who supported this research direction. Amir Yazdanbakhsh and Deniz Altinbüken contributed to this paper in an advisory capacity.

This work was supported in part by the National Science Foundation (NSF) under Grant #CCF-2453323 and a Google Academic Research Award.

References

Abdi, J., Zhang, G., and Jeffrey, M. C. Brief announcement: Is the problem-based benchmark suite fearless with rust? In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23)*. ACM, 2023a. doi: 10.1145/3558481.3591313.

Abdi, J., Zhang, G., and Jeffrey, M. C. Rusty-PBBS: Rust problem based benchmark suite. https://github.com/mcj-group/rpb, 2023b. GitHub repository.

Ahmed, T. and Devanbu, P. Learning code summarization from a small and local dataset, 2022. URL https://arxiv.org/abs/2206.00804.

Anderson, D., Blelloch, G., Dhulipala, L., Tseng, T., wheatman, Hübschle, L., Yesantharao, R., Dong, X., and aheydon google. Parlaylib: A toolkit for programming parallel algorithms on shared-memory multicore machines (github repository; cmu parlay group), 2020. URL https://github.com/cmuparlay/parlaylib.

Anderson, D., Blelloch, G. E., Dhulipala, L., Dobson, M., and Sun, Y. The problem-based benchmark suite (PBBS), V2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. ACM, 2022. doi: 10.1145/3503221.3508422.

Assumpção, H., Ferreira, D., Campos, L., and Murai, F. Codeevolve: an open source evolutionary coding agent for algorithm discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

Aygün, E., Belyaeva, A., Comanici, G., Coram, M., Cui, H., Garrison, J., Johnston, R., Kast, A., McLean, C. Y., Norgaard, P., Shamsi, Z., Smalling, D., Thompson, J., Venugopalan, S., Williams, B. P., He, C., Martinson, S., Plomecka, M., Wei, L., Zhou, Y., Zhu, Q.-Z., Abraham, M., Brand, E., Bulanova, A., Cardille, J. A., Co, C., Ellsworth, S., Joseph, G., Kane, M., Krueger, R., Kartiwa, J., Liebling, D., Lueckmann, J.-M., Raccuglia, P., Wang, X. J., Chou, K., Manyika, J., Matias, Y., Platt, J. C., Dorfman, L., Mourad, S., and Brenner, M. P. An ai system to help scientists write expert-level empirical software. *arXiv preprint arXiv:2509.06503*, 2025. doi: 10.48550/ARXIV.2509.06503. URL https://arxiv.org/abs/2509.06503.

Bitan, T., Kadosh, T., Kaplan, E., Meiri, S., Chen, L., Morales, P., Hasabnis, N., and Oren, G. Unipar: A unified llm-based framework for parallel and accelerated code translation in hpc. In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2025.

Blelloch, G. E., Anderson, D., and Dhulipala, L. ParlayLib: A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, pp. 507–509, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369350. doi: 10.1145/3350755.3400254. URL https://doi.org/10.1145/3350755.3400254.

Blumofe, R. D. and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999. doi: 10.1145/324133.324234.

Brent, R. P. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pp. 49–60, USA, 2013. USENIX Association.

Chaturvedi, A. Hpccoder-v2: Efficient fine-tuning of small language models for high-performance computing. *arXiv preprint arXiv:2410.20527*, 2024.

Chen, L., Bhattacharjee, A., Ahmed, N., Hasabnis, N., Oren, G., Vo, V., and Jannesari, A. *OMPGPT: A Generative Pre-trained Transformer Model for OpenMP*, pp. 121–134. Springer Nature Switzerland, 2024. ISBN 9783031695773. doi: 10.1007/978-3-031-69577-3_9. URL http://dx.doi.org/10.1007/978-3-031-69577-3_9.

CodeRosetta. Coderosetta/coderosetta_cpp_cuda_base: base c++ to cuda translation model (hugging face), 2024. URL https://huggingface.co/CodeRosetta/CodeRosetta_cpp_cuda_base.

Davis, J. H., Nichols, D., Khillan, I., and Bhatele, A. Pareval-repo: A benchmark suite for evaluating llms with repository-level hpc translation tasks. In *Proceedings of the International Conference on Parallel Processing (ICPP 2025)*, 2025a. doi: 10.1145/3754598.3754669. URL https://doi.org/10.1145/3754598.3754669.

Davis, J. H., Nichols, D., Khillan, I., and Bhatele, A. Pareval-repo: A benchmark suite for evaluating llms with repository-level hpc translation tasks. *arXiv preprint arXiv:2506.20938*, 2025b. doi: 10.48550/ARXIV.2506.20938. URL https://arxiv.org/abs/2506.20938.

DMOJ Developers. DMOJ: Modern online judge. https://github.com/DMOJ/online-judge, 2024. GitHub repository.

Du, M., Tuan, L. A., Liu, Y., Qing, Y., Huang, D., He, X., Liu, Q., Ma, Z., and kiong Ng, S. Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization. *arXiv preprint arXiv:2505.23387*, 2025.

Eniser, H. F., Zhang, H., David, C., Wang, M., Christakis, M., Paulsen, B., Dodds, J., and Kroening, D. Towards translating real-world code with llms: A study of translating to rust, 2024. URL https://arxiv.org/abs/2405.11514.

Foundry, P. C., Nichols, D., Zi, Y., Xie, Z., and Menon, H. Pareval: Parallel code evaluation benchmark (github repository), 2024. URL https://github.com/parallelcodefoundry/ParEval.

GraphIt-DSL, Zhang, Y., Manlaibaatar, T., Brahmakshatriya, A., ykenny1, Baghdadi, R., Furst, E., Siegfried, G., Wade, B., and Dhulipala, L. Graphit-dsl/graphit: Graphit compiler and dsl implementation (github repository), 2018. URL https://github.com/GraphIt-DSL/graphit.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2022. URL https://openreview.net/pdf?id=nZeVKeeFYf9.

Husein, R. A., Aburajouh, H., and Catal, C. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92:103917, 2025. ISSN 0920-5489. doi: https://doi.org/10.1016/j.csi.2024.103917. URL https://www.sciencedirect.com/science/article/pii/S0920548924000862.

Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L., and Murthy, A. Llms can't plan, but can help planning in llm-modulo frameworks, 2024. URL https://arxiv.org/abs/2402.01817.

kcxain. kcxain/translator-qwen3-0.6b: Musl c-to-cuda translator model (hugging face), 2025. URL https://huggingface.co/kcxain/translator-Qwen3-0.6B.

Ke, C. kcxain/musl: code repository for mutual-supervised learning for sequential-to-parallel code translation, 2025. URL https://github.com/kcxain/musl.

Ke, C., Zhang, R., Wang, S., Ding, L., Li, G., Wen, Y., Zhang, S., Xu, R., Qin, J., Guo, J., Wang, C., Li, L., Guo, Q., and Chen, Y. Mutual-supervised learning for sequential-to-parallel code translation. *arXiv preprint arXiv:2506.11153*, 2025. doi: 10.48550/ARXIV.2506.11153. URL https://arxiv.org/abs/2506.11153.

Khrulkov, V., Galichin, A., Bashkirov, D., Vinichenko, D., Travkin, O., Alferov, R., Kuznetsov, A., and Oseledets, I. Gigaevo: An open source optimization framework powered by llms and evolution algorithms. *arXiv preprint arXiv:2511.17592*, 2025.

Lei, K., Yang, H., Zhang, H., You, X., Zhang, K., Luan, Z., Liu, Y., and Qian, D. Pragma: A profiling-reasoned multi-agent framework for automatic kernel optimization. *arXiv preprint arXiv:2511.06345*, 2025.

Mahmud, Q. I., TehraniJamsaz, A., Phan, H. D., Ahmed, N. K., and Jannesari, A. Autoparllm: Gnn-guided automatic code parallelization using large language models, 2023. URL https://arxiv.org/abs/2310.04047.

Merouani, M., Bernou, I. K., and Baghdadi, R. Agentic auto-scheduling: An experimental study of llm-guided loop optimization. *arXiv preprint arXiv:2511.00592*, 2025.

Nichols, D., Davis, J. H., Xie, Z., Rajaram, A., and Bhatele, A. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24)*. ACM, 2024. doi: 10.1145/3625549.3658689. URL https://pssg.cs.umd.edu/assets/papers/2024-06-pareval-hpdc.pdf.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

ParEVO. Parevo project repository (code/data for parevo; includes parlay-instruct artifacts), 2026a. URL https://github.com/WildAlg/ParEVO.

ParEVO. Deepseek-parlay-6.7b: A fine-tuned model for parallel algorithmic reasoning, 2026b. URL https://huggingface.co/qqggez/deepseek-parlay-6.7b.

ParEVO. Qwen3-30b-sft-stage2-merged, 2026. URL https://huggingface.co/qqggez/qwen3-30b-sft-stage2-merged.

ParEVO. Qwen3-rust-dpo: A fine-tuned model for safe parallel rust, 2026. URL https://huggingface.co/YangLiuWillow/qwen3_rust_dpo_final_merged.

Press, O., Amos, B., Zhao, H., Wu, Y., Ainsworth, S. K., Krupke, D., Kidger, P., Sajed, T., Stellato, B., Park, J., Bosch, N., Meril, E., Steppi, A., Zharmagambetov, A., Zhang, F., Perez-Pineiro, D., Mercurio, A., Zhan, N., Abramovich, T., Lieret, K., Zhang, H., Huang, S., Bethge, M., and Press, O. Algotune: Can language models speed up general-purpose numerical programs? *arXiv preprint arXiv:2507.15887*, 2025a. doi: 10.48550/ARXIV.2507.15887. URL https://arxiv.org/abs/2507.15887.

Press, O., Amos, B., Zhao, H., Wu, Y., Ainsworth, S. K., Krupke, D., Kidger, P., Sajed, T., Stellato, B., Park, J., Bosch, N., Meril, E., Steppi, A., Zharmagambetov, A., Zhang, F., Perez-Pineiro, D., Mercurio, A., Zhan, N., Abramovich, T., Lieret, K., Zhang, H., Huang, S., Bethge, M., and Press, O. Algotune benchmark dataset, 2025b. URL https://huggingface.co/datasets/oripress/AlgoTune.

Rahman, M. Marco: Multi-agent reasoning for code optimization. *arXiv preprint arXiv:2501.12345*, 2025.

Ren, S., Hu, Q., He, Y., Li, G., Lu, L., Liu, D., Jin, Z., and Lyu, M. R. CodeBLEU: a method for automatic evaluation of code synthesis. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020.

Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., and Özsu, M. T. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal*, 29(2–3):595–618, June 2019. ISSN 0949-877X. doi: 10.1007/s00778-019-00548-x. URL http://dx.doi.org/10.1007/s00778-019-00548-x.

Sharma, A. Openevolve: an open-source evolutionary coding agent, 2025a. URL https://github.com/algorithmicsuperintelligence/openevolve.

Sharma, A. Openevolve: An open source implementation of google deepmind's alphaevolve. https://github.com/codelion/openevolve, 2025b. Accessed: 2026-01-24.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

Shun, J. and Blelloch, G. E. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 135–146. ACM, 2013. doi: 10.1145/2442516.2442530. URL https://doi.org/10.1145/2442516.2442530.

Shun, J., Blelloch, G. E., Kyrola, A., Simhadri, H. V., Tangwongsan, K., Fineman, J. T., and Gibbons, P. B. Brief announcement: The problem based benchmark suite. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. ACM, 2012. doi: 10.1145/2312005.2312018.

Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J., Hashemi, M., Neubig, G., Ranganathan, P., Bastani, O., and Yazdanbakhsh, A. Learning performance-improving code edits, 2024. URL https://arxiv.org/abs/2302.07867.

Singh, G., Guha, A., Kailkhura, B., and Menon, H. Can test-time compute help LLMs write low-resource parallel code better? In *NeurIPS Workshop on Deep Learning for Code (DL4C)*, 2024. URL https://openreview.net/forum?id=0RnJzt8v84.

Surina, A., Mansouri, A., Quaedvlieg, L., Seddas, A., Viazovska, M., Abbe, E., and Gulcehre, C. Algorithm discovery with llms: Evolutionary search meets reinforcement learning. *CoRR*, abs/2504.05108, 2025. doi: 10.48550/ARXIV.2504.05108. URL https://doi.org/10.48550/arXiv.2504.05108.

TehraniJamsaz, A., Bhattacharjee, A., Chen, L., Ahmed, N. K., Yazdanbakhsh, A., and Jannesari, A. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Wen, Y., Guo, Q., Fu, Q., Li, X., Xu, J., Tang, Y., Zhao, Y., Hu, X., Du, Z., Li, L., Wang, C., Zhou, X., and Chen, Y. Babeltower: Learning to auto-parallelized program translation. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 23685–23700. PMLR, 2022. doi: unspecified. URL https://proceedings.mlr.press/v162/wen22b.html.

Yang, Z. Perfcoder: Performance-driven code generation. *arXiv preprint arXiv:2502.54321*, 2025.

Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., and Amarasinghe, S. P. Graphit: a high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):121:1–121:30, 2018. doi: 10.1145/3276491. URL https://doi.org/10.1145/3276491.

A. Evolutionary Coding Agent (ECA) System Prompt

The structural system prompt we specify for the single ECA node is as follows:

ECA System Prompt

```
You are an expert C++ competitive programmer. Your task is to write a
COMPLETE, CORRECT, and FAST C++ solution.

PROBLEM:
{problem_description}

REQUIREMENTS:
Write a complete C++ parallel program that compiles and runs correctly
Read input from standard input (cin)
Write output to standard output (cout)
Handle all edge cases mentioned in the problem
Optimize for speed - use efficient algorithms and data structures
Use C++ STL where appropriate (vector, map, set, priority_queue, etc.)
Consider time complexity and space complexity
The parlay library MUST be used as the core computation of the program

AVAILABLE LIBRARIES:
Standard C++ libraries (iostream, algorithm, vector, map, etc.)
The parlay library

Note:
parlay::parallel_for does not guarantee ordering, do not use it with IO operations.

CODE STYLE:
Use C++ style comments: // for single line, /* */ for multi-line
Do NOT use Python-style # comments
Comments should be simple and short
Include necessary headers
Write clean, readable code

OUTPUT FORMAT:
Return ONLY the complete C++ code. Do not include explanations,
markdown formatting, or code blocks.
Just the raw C++ source code that can be directly compiled.
```

B. Detailed Experimental Data

B.1. Comprehensive ParEval Benchmarks

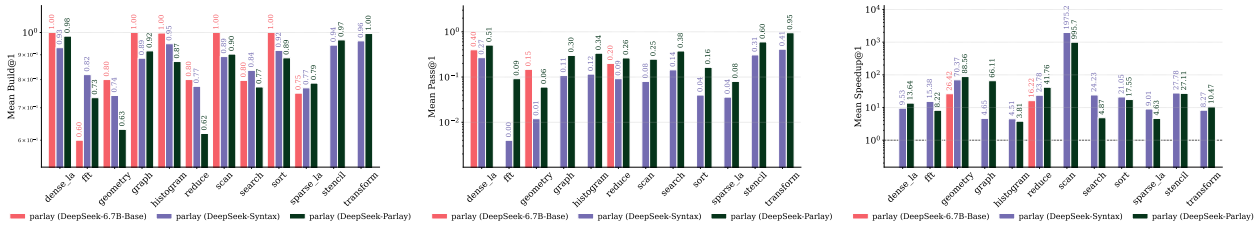
Table 5 provides the complete breakdown of ‘Build@1‘, ‘Pass@1‘, and ‘Speedup@1‘ metrics across commercial and open-weight models. The fine-tuned ParEVO models consistently outperform baselines in compilation rates and execution speed.

Model	Temp.	Code	Sched.	Build@1	Pass@1	Speedup
Gemini-2.5-Flash	0.2	Parlay	Parlay	0.58	0.29	13.42
Gemini-2.5-Pro	0.2	Parlay	Parlay	0.98	0.77	10.40
Gemini-3-Pro	0.2	Parlay	Parlay	0.25	0.23	12.29
GPT-5 Thinking	0.2	Parlay	Parlay	0.73	0.63	14.03
Claude Opus 4.5	0.2	Parlay	Parlay	0.28	0.27	0.65
DeepSeek-Parlay (ParEVO)	0.2	**Parlay**	**Parlay**	**0.79**	**0.35**	**16.40**
Gemini-2.5-Parlay (ParEVO)	0.2	**Parlay**	**Parlay**	**0.84**	**0.33**	**106.87**
Qwen3-Parlay (ParEVO)	0.2	**Parlay**	**Parlay**	**0.50**	**0.33**	**8.63**
DeepSeek-Syntax	0.2	Parlay	Parlay	0.85	0.12	6.60
Qwen2.5-Coder-32B	0.2	Parlay	Parlay	0.93	0.11	9.98
Qwen2.5-Coder-32B	0.7	Parlay	Parlay	0.86	0.17	15.85
Qwen2.5-Coder-32B-Instruct	0.2	Parlay	Parlay	0.61	0.41	12.91
Qwen3-Coder-30B-Instruct	0.2	Parlay	Parlay	0.51	0.28	8.61
DeepSeek-6.7B-Base	0.2	Parlay	Parlay	0.89	0.11	3.65
DeepSeek-Coder-V2-Lite-Base	0.2	Parlay	Parlay	0.80	0.09	2.57
DeepSeek-Coder-V2-Lite-Base	0.7	Parlay	Parlay	0.92	0.14	6.79
StarCoder2-15B	0.2	Parlay	Parlay	0.80	0.27	20.20
StarCoder2-15B	0.7	Parlay	Parlay	0.81	0.15	37.75
Gemini-3-Pro	0.2	OMP	OMP	0.78	0.72	23.13
Gemini-2.5-Parlay	0.2	OMP	OMP	0.94	0.71	23.84
Qwen2.5-Coder-32B-Instruct	0.2	OMP	OMP	0.91	0.65	13.36
Qwen2.5-Coder-32B-Instruct	0.7	OMP	OMP	0.92	0.65	14.43
Qwen3-Coder-30B-Instruct	0.2	OMP	OMP	0.86	0.55	16.53
Qwen3-Coder-30B-Instruct	0.7	OMP	OMP	0.91	0.56	15.17
Qwen2.5-Coder-32B	0.2	OMP	OMP	0.98	0.35	15.61
Qwen2.5-Coder-32B	0.7	OMP	OMP	0.97	0.39	12.86
DeepSeek-Coder-V2-Lite-Base	0.2	OMP	OMP	0.82	0.24	8.86
DeepSeek-Coder-V2-Lite-Base	0.7	OMP	OMP	0.96	0.39	16.06
StarCoder2-15B	0.2	OMP	OMP	0.97	0.26	12.32
StarCoder2-15B	0.7	OMP	OMP	0.95	0.30	11.7
Gemini-3-Pro	0.2	Rust	Rayon	0.97	0.82	7.42
Qwen2.5-Coder-32B-Instruct	0.2	Rust	Rayon	0.63	0.49	5.97
Qwen2.5-Coder-32B-Instruct	0.7	Rust	Rayon	0.70	0.48	6.55
Qwen3-Rust (ParEVO)	0.2	**Rust**	**Rayon**	**0.64**	**0.46**	**6.10**
Qwen3-Coder-30B-Instruct	0.2	Rust	Rayon	0.61	0.50	5.70
Qwen3-Coder-30B-Instruct	0.7	Rust	Rayon	0.66	0.49	5.64
Qwen2.5-Coder-32B	0.2	Rust	Rayon	0.82	0.45	5.64
Qwen2.5-Coder-32B	0.7	Rust	Rayon	0.86	0.38	4.48
DS-Coder-V2-Lite-Base	0.2	Rust	Rayon	0.73	0.29	6.26
DS-Coder-V2-Lite-Base	0.7	Rust	Rayon	0.85	0.25	5.21
StarCoder2-15B	0.2	Rust	Rayon	0.77	0.27	3.58
StarCoder2-15B	0.7	Rust	Rayon	0.82	0.25	5.66
DS-Coder-V2-Lite-Instruct	0.2	Rust	Rayon	0.40	0.02	0.77
DS-Coder-V2-Lite-Instruct	0.7	Rust	Rayon	0.48	0.02	0.76

Table 5. Comprehensive ParEval results for commercial and local models. Shaded regions distinguish C++/ParlayLib/OMP models (Green) from Rust/Rayon models (Purple).

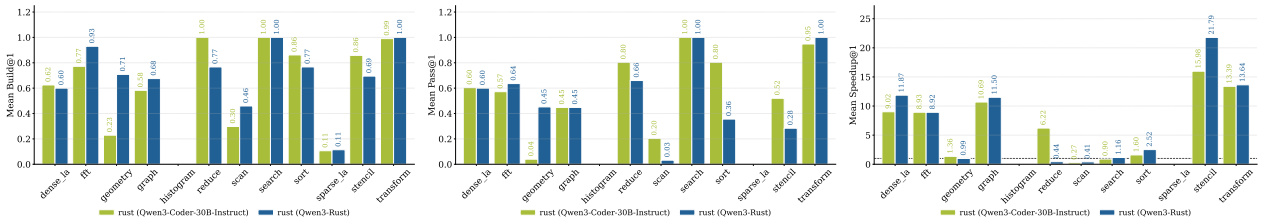
B.2. Metric Breakdown by Problem Type

To understand the specific impact of fine-tuning, we visualize the shift in metrics across problem types. Figure 7 and Figure 4 demonstrate that while fine-tuning universally improves `Build` and `Pass` rates, the `Speedup` gains are most pronounced in the irregular graph and complex arithmetic categories.



(a) Mean Build@1 *(b)* Mean Pass@1 *(c)* Mean Speedup@1

Figure 7. Impact of Fine-tuning on DeepSeek-6.7B. The fine-tuned model (DeepSeek-Parlay) shows massive gains in pass rate and speedup compared to the base model. The DeepSeek-Syntax model is the finetuned model of DeepSeek-6.7B-Base purely on ParlayLib syntax.



(a) Mean Build@1 *(b)* Mean Pass@1 *(c)* Mean Speedup@1

Figure 8. Impact of Fine-tuning on Qwen3-Coder-30B-A3B-Instruct. The fine-tuned model (Qwen3-Rust) shows gains in speedup compared to the base model.

B.3. Comparison vs. Expert Human Baselines (PBBS & RPB)

A key contribution of this work is benchmarking against expert human code. Figure 9 compares our best generated solutions against the PBBSBench (C++) and RPB (Rust) baselines. ParEVO solutions frequently match or exceed the human baselines. Figure 9 visualizes the runtime and scalability profiles.

B.4. Case Study: The Safety vs. Performance Trade-off

A deeper analysis of the Graph Shortest Path problem reveals a subtle trade-off introduced by fine-tuning. As shown in Figure 15, the base model produces a wide variance of runtimes, occasionally hitting a very fast (but risky) solution using atomic operations. The fine-tuned ParEVO model produces highly consistent but slightly slower code, preferring safe high-level primitives (like 'parlay::unique') over raw memory manipulation. The detailed code samples are shown in Figure 16.

B.5. Case Study: Performance Stability on ParEval Problem 34 (Scan)

Similarly to the Shortest Path problem, we observe a distinct stabilization of performance in the fine-tuned model for ParEval Problem 34 (Scan), as shown in Figure 17. The base model's runtime distribution Figure 17(a) is somewhat disjointed, with some runs being very slow and others faster. On the other hand, the fine-tuned model Figure 17(b) demonstrates a much tighter, more predictable runtime distribution. This consistency confirms that the fine-tuned ParEVO model systematically converges on stable and reliable parallel patterns.

B.6. Failure Modes: Geometric Hallucinations

While fine-tuning improves general syntax, it can induce "confident hallucinations" in domains with specialized logic. In the Convex Hull task (Table 6), the fine-tuned model failed by repeatedly calling a non-existent `parlay::convex_hull` function, whereas the base model attempted (and occasionally succeeded at) a manual implementation. This highlights the necessity of the ECA's compiler-feedback loop to catch API hallucinations.

Problem Type	Model	Pass@1	Speedup@1
10_convex_hull	Gemini-2.5-Pro	0.45	1.43
10_convex_hull	**DS-Parlay (ParEVO)**	**0.00**	**0.00**
13_closest_pair_2d	Gemini-2.5-Pro	0.40	74.48
13_closest_pair_2d	**DS-Parlay (ParEVO)**	**0.45**	**188.03**

Table 6. Detailed Geometry Results. The fine-tuned model dominates in Closest Pair but hallucinates APIs in Convex Hull.

C. Prompts

ParEval Prompts For the ParEval benchmarks, we adopt the prompting specifications outlined by Nichols et al. (Nichols et al., 2024). We utilize a fixed system instruction alongside language-specific templates for C++ and Rust.

The system prompt provided to the model is as follows (see Figure 18).

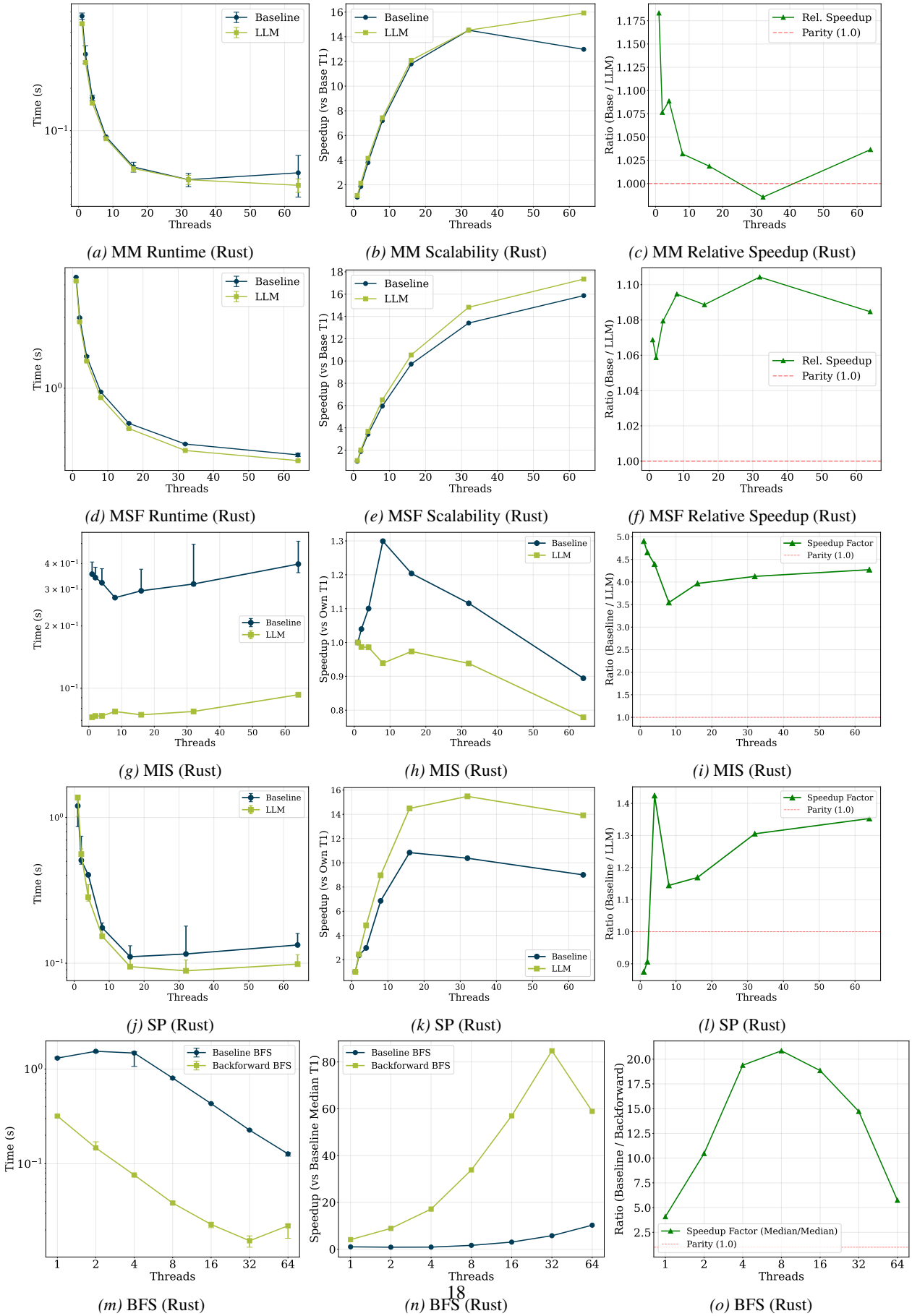
Extending ParEval for Parallel Libraries Since the original ParEval dataset lacks native support for ParlayLib and Rust, we manually curated task-specific prompts to bridge this gap. These prompts preserve the original problem semantics while explicitly requesting the use of specific parallel frameworks (ParlayLib for C++ and Rayon for Rust). Figure 19 demonstrates how a standard Discrete Fourier Transform (DFT) task is adapted for both languages.

PBBSBench Prompting Strategy We employ two distinct prompting strategies for PBBSBench to evaluate the model's ability to utilize context:

- **Concise Prompts:** These contain only the natural language problem description and the target function signature.
- **Augmented Prompts:** These extend the concise version by including definitions for necessary ParlayLib primitives, custom data structures (e.g., `Graph`), and helper utilities (e.g., `Graph_io`) defined within the PBBSBench environment.

Figure 20 illustrates an example of the concise prompting format.

RPB Prompting Strategy The prompting strategy for the RPB benchmarks relies on a composite structure. Each prompt comprises two distinct segments: (1) a context block defining Rust primitives that replicate ParlayLib functionality (e.g., `flatten`), and (2) the specific problem statement, including allowed libraries and the target function signature.



(a) MM Runtime (Rust) *(b)* MM Scalability (Rust) *(c)* MM Relative Speedup (Rust)

(d) MSF Runtime (Rust) *(e)* MSF Scalability (Rust) *(f)* MSF Relative Speedup (Rust)

(g) MIS (Rust) *(h)* MIS (Rust) *(i)* MIS (Rust)

(j) SP (Rust) *(k)* SP (Rust) *(l)* SP (Rust)

(m) BFS (Rust) *(n)* BFS (Rust) *(o)* BFS (Rust)

Figure 9. Runtime and Scalability comparisons against expert Rust and C++ baselines. ParEVO solutions track or beat the scalability of hand-optimized code. In (m)-(o), *BackForward BFS* specifically refers to the new BFS algorithm ParEVO generated, which uses a different method than the baseline implementation that uses multiqueue BFS.

RPB (Unsafe Baseline)

```rust
#[cfg(not(feature = "openevolve"))]
pub fn maximal_matching(ea: &EdgeArray) -> Vec<DefInt> {
    let n = std::cmp::max(ea.num_rows, ea.num_cols);
    // ... setup ...
    let matched: Vec<bool> = (0..n).into_par_iter().map(|_| false).collect();
    let matched_ptr = matched.as_ptr() as usize;

    let reserve = |i: usize| -> bool {
        let (u, v) = (ea[i].u as usize, ea[i].v as usize);
        // MIXED ACCESS: Reading safe memory, writing via raw pointer later.
        // Compiler may optimize this read incorrectly due to aliasing.
        if matched[u] || matched[v] || u == v { false }
        else {
            rs[u].reserve(i as u32); rs[v].reserve(i as u32);
            true
        }
    };

    let commit = |i: usize| -> bool {
        // ... (check logic) ...
        if rs[v].check(i as u32) {
            rs[v].reset();
            if rs[u].check(i as u32) {
                unsafe { // UNSAFE WRITE via pointer alias
                    (matched_ptr as *mut bool).add(u).write(true);
                    (matched_ptr as *mut bool).add(v).write(true);
                }
                return true;
            }
        }
        // ...
    };

    (0..m).spec_for(reserve, commit, 10, ...).unwrap();

    // DOUBLE PASS (Inefficient collection)
    let mut matching_idx = vec![];
    parlay::primitives::pack(
        &rs.par_iter().map(|r| r.get() as DefInt).collect::<Vec<DefInt>>(),
        &rs.par_iter().map(|r| r.reserved()).collect::<Vec<bool>>(),
        &mut matching_idx
    );
    matching_idx
}
```

ParEVO (Optimized)

```rust
#[cfg(feature = "openevolve")]
pub fn maximal_matching(ea: &EdgeArray) -> Vec<DefInt> {
    let n = std::cmp::max(ea.num_rows, ea.num_cols);
    // ... setup ...
    let matched: Vec<bool> = (0..n).into_par_iter().map(|_| false).collect();
    let matched_ptr = matched.as_ptr() as usize;

    let reserve = |i: usize| -> bool {
        let (u, v) = (ea[i].u as usize, ea[i].v as usize);
        if u == v { return false; }

        // CONSISTENT RAW READ: Forces fresh memory access.
        unsafe {
            let mp = matched_ptr as *const bool;
            if *mp.add(u) || *mp.add(v) { return false; }
        }
        rs[u].reserve(i as u32); rs[v].reserve(i as u32);
        true
    };

    let commit = |i: usize| -> bool {
        // ... (check logic) ...
        if rs[v].check(i as u32) {
            rs[v].reset();
            if rs[u].check(i as u32) {
                unsafe { // UNSAFE WRITE
                    let mp = matched_ptr as *mut bool;
                    *mp.add(u) = true; *mp.add(v) = true;
                }
                return true;
            }
        }
        // ...
    };

    // TUNED BLOCK SIZE
    (0..m).spec_for(reserve, commit, 16, ...).unwrap();

    // SINGLE PASS (Optimized collection)
    let mut matching_idx = Vec::with_capacity(n / 2);
    let (values, flags): (Vec<DefInt>, Vec<bool>) = rs
        .par_iter()
        .map(|r| (r.get() as DefInt, r.reserved()))
        .unzip();

    parlay::primitives::pack(&values, &flags, &mut matching_idx);
    matching_idx
}
```

Figure 10. Code comparison for Maximal Matching. **Left (Baseline):** Uses mixed safe/unsafe access (potential aliasing bugs) and collects results using two separate passes (red highlights). **Right (ParEVO):** Uses consistent raw pointer access to ensure memory visibility, increases block granularity to 16, and uses a single-pass `unzip` for result collection (green highlights).

RPB (Unsafe Baseline)

```rust
#[cfg(not(feature="openevolve"))]
pub fn minimum_spanning_forest(wea: &WghEdgeArray, dest: &mut
    Vec<DefInt>) {
    // ... Initialization ...
    let rs: Vec<Reservation> = (0..n).map(|_| Reservation::
      new()).collect();

    // Raw pointers for some structures only
    let _uf_ptr = &uf as *const _ as usize;
    let _iwea_ptr = iwea.as_ptr() as usize;

    let reserve = |i: usize| {
        // Unsafe access to Edge List
        let e = unsafe {
            (_iwea_ptr as *mut IndexedEdge).add(i).as_mut()
      .unwrap() // Runtime check
        };
        // Unsafe access to UnionFind
        let luf = unsafe { (_uf_ptr as *mut UnionFind).as_mut
      () .unwrap() };

        e.u = luf.find(e.u as DefIntS) as DefInt;
        e.v = luf.find(e.v as DefIntS) as DefInt;

        if e.u != e.v {
            // STANDARD INDEXING: Incurs bounds checking
      overhead
            rs[e.v as usize] .reserve(i as DefInt);
            rs[e.u as usize] .reserve(i as DefInt);
            true
        } else { false }
    };

    // ... Commit logic similar to above ...
    (0..iwea.len()).spec_for(reserve, commit, ...);
}
```

ParEVO (Optimized)

```rust
#[cfg(feature="openevolve")]
pub fn minimum_spanning_forest(wea: &WghEdgeArray, dest: &mut
    Vec<DefInt>) {
    // ... Initialization ...
    let rs: Vec<Reservation> = (0..n).into_par_iter().map(|_|
      Reservation::new()).collect();

    // Raw pointers for EVERYTHING (including Reservation
      array)
    let _rs_ptr = rs.as_ptr() as usize;
    let _uf_ptr = &uf as *const _ as usize;
    let _iwea_ptr = iwea.as_ptr() as usize;

    let reserve = |i: usize| {
        unsafe {
            // UNCHECKED UNWRAP: Eliminates null checks
            let e = (_iwea_ptr as *mut IndexedEdge).add(i)
                .as_mut() .unwrap_unchecked() ;
            let luf = (_uf_ptr as *mut UnionFind).as_mut()
                .unwrap_unchecked() ;

            e.u = luf.find(e.u as DefIntS) as DefInt;
            e.v = luf.find(e.v as DefIntS) as DefInt;

            if e.u != e.v {
                // POINTER ARITHMETIC: Eliminates bounds
      checks
                let rv = (_rs_ptr as *const Reservation)
      .add(e.v as usize)
                    .as_ref().unwrap_unchecked();
                let ru = (_rs_ptr as *const Reservation)
      .add(e.u as usize)
                    .as_ref().unwrap_unchecked();

                rv.reserve(i as DefInt);
                ru.reserve(i as DefInt);
            }
            true
        }
    };
    (0..iwea.len()).spec_for(reserve, commit, ...);
}
```

Figure 11. Code comparison for Minimum Spanning Forest (MSF). **Left (Baseline):** Uses standard indexing for the reservation array (incurring bounds checks) and standard `unwrap()` (incurring branch checks), highlighted in red. **Right (ParEVO):** Adopts a "Maximal Unsafe" strategy, converting all data structures to raw pointers. It uses `unwrap_unchecked()` and pointer arithmetic (`.add()`) to eliminate all runtime safety checks, highlighted in green. This relies on the assumption that edge indices are always valid, allowing ParEVO to trade runtime safety checks for improved performance.

RPB (Unsafe Baseline)

```rust
#[cfg(not(feature = "openevolve"))]
pub fn maximal_independent_set(g: &Graph) -> Vec<u8> {
    let n = g.n;
    // UNSAFE: Standard Vec used for concurrent access
    let flags: Vec<u8> = (0..n).into_par_iter()
        .map(|_| 0) .collect() ; // Standard allocation
    let flags_ptr = flags.as_ptr() as usize;

    let reserve = |i: usize, s: &mut MISState| -> bool {
        s.flag = 1;
        let v = g.index(i);
        for j in 0..v.degree {
            let ngh = v.neighbors[j] as usize;
            if ngh < i {
                // DATA RACE: Reading mutable memory without atomics
                // Compiler may optimize incorrectly; Undefined Behavior
                let f = flags[ngh] ;
                if f == 1 { s.flag = 2; return true; }
                else if f == 0 { s.flag = 0; }
            }
        }
        true
    };

    let commit = |i: usize, s: &mut MISState| -> bool {
        // UNSAFE POINTER WRITE: Bypassing borrow checker
        unsafe { (flags_ptr as *mut u8).add(i)
      .write(s.flag) ; }
        s.flag > 0
    };

    (0..n).stateful_spec_for(
        reserve, commit, MISState { flag: 0 },
        20 , Some(64), Some(256) // Small granularity
    ).expect("failed speculative for");

    return flags;
}
```

ParEVO (Optimized)

```rust
#[cfg(feature = "openevolve")]
pub fn maximal_independent_set(g: &Graph) -> Vec<u8> {
    let n = g.n;
    // PARALLEL ATOMICS: Safe concurrent access
    let flags: Vec<AtomicU8> = (0..n).into_par_iter()
        .map(|_| AtomicU8::new(0)) .collect() ;
    let flags_slice = &flags[..];

    let reserve = |i: usize, s: &mut MISState| -> bool {
        let v = g.index(i);
        let mut waiting = false;
        for &ngh in v.neighbors {
            let ngh = ngh as usize;
            if ngh < i {
                // SAFE ATOMIC LOAD: Correct synchronization
                let f = unsafe {
                    flags_slice.get_unchecked(ngh)
      .load(Relaxed)
                };
                if f == 1 { s.flag = 2; return true; }
                if f == 0 { waiting = true; }
            }
        }
        s.flag = if waiting { 0 } else { 1 };
        true
    };

    let commit = |i: usize, s: &mut MISState| -> bool {
        if s.flag > 0 {
            // SAFE ATOMIC STORE
            unsafe { flags_slice.get_unchecked(i)
      .store(s.flag, Relaxed) ; }
            true
        } else { false }
    };

    (0..n).stateful_spec_for(
        reserve, commit, MISState { flag: 0 },
        256 , None, None // Larger granularity
    ).expect("failed speculative for");

    // ZERO-COPY TRANSFORMATION: AtomicU8 -> u8
    unsafe {
        let mut v = std::mem::ManuallyDrop::new(flags);
        Vec::from_raw_parts (v.as_mut_ptr() as *mut u8, v.len
  (), v.capacity())
    }
}
```

Figure 12. Code comparison for Maximal Independent Set (MIS). **Left (Baseline):** Uses unsafe standard `Vec<u8>` (red), causing undefined behavior (data races) during reads and writing via raw pointers. It uses a small block size (20). **Right (ParEVO):** Uses `Vec<AtomicU8>` (green) for correct synchronization using `Relaxed` ordering. It optimizes throughput with a larger block size (256) and employs a zero-copy cast to convert the atomic vector back to a standard vector at the end.

RPB (Unsafe Baseline)

```rust
#[cfg(not(feature = "openevolve"))]
pub fn spanning_forest(ea: &EdgeArray) -> Vec<u32> {
    let n = ea.num_rows;
    // NON-ATOMIC & FRESH ALLOCATION
    let uf = UnionFind::new(n);
    let uf_ptr = &uf as *const UnionFind as usize;

    // HEAVY ALLOCATION: Creates new Vec every call
    let rs: Vec<Reservation> = (0..n).into_par_iter()
        .map(|_| Reservation::new()) .collect() ;

    let reserve = |i: usize, s: &mut SFState| -> bool {
        let e = &ea[i]; // Bounds checked
        unsafe {
            // UNSAFE DEREF + RUNTIME CHECK (unwrap)
            s.u = (uf_ptr as *mut UnionFind).as_mut()
     .unwrap() .find(e.u as i32);
            s.v = (uf_ptr as *mut UnionFind).as_mut()
     .unwrap() .find(e.v as i32);
        }
        if s.u > s.v { swap(&mut s.u, &mut s.v); }

        if s.u != s.v {
            // BOUNDS CHECKED indexing
            rs[s.v as usize] .reserve(i as DefInt);
            true
        } else { false }
    };

    let commit = |i: usize, s: &mut SFState| -> bool {
        if rs[s.v as usize].check(i as DefInt) {
            unsafe {
                (uf_ptr as *mut UnionFind).as_mut().unwrap().
  link(s.v, s.u);
            }
            true
        } else { false }
    };

    (0..ea.non_zeros).stateful_spec_for(
        reserve, commit, SFState { u: -1, v: -1 },
        100, Some(1024), Some(4096)
    ).expect("failed speculative for");

    rs.into_par_iter().filter_map(|r| /*...*/).collect()
}
```

ParEVO (Optimized)

```rust
#[cfg(feature = "openevolve")]
pub fn spanning_forest(ea: &EdgeArray, rs_cache: &mut Option<
    Vec<Reservation>>) -> Vec<u32> {
    let n = ea.num_rows;
    let uf = AtomicUnionFind::new(n); // Safe Atomics

    // MEMORY RECYCLING: Reuses vector to skip allocation
    let mut rs = if let Some(mut vec) = rs_cache.take() {
        if vec.len() == n {
            vec.par_iter_mut().for_each(|r| *r = Reservation
  ::new());
            vec
        } else { (0..n).into_par_iter().map(|_| Reservation::
  new()).collect() }
    } else { (0..n).into_par_iter().map(|_| Reservation::new
  ()).collect() };

    let es = &ea.es;
    let reserve = |i: usize, s: &mut SFState| -> bool {
        // ZERO OVERHEAD ACCESS
        let e = unsafe { es.get_unchecked(i) };
        if e.u == e.v { return false; }

        let u_val = uf.find(e.u as i32);
        let v_val = uf.find(e.v as i32);
        if u_val == v_val { return false; }

        let (u, v) = if u_val > v_val { (v_val, u_val) } else
      { (u_val, v_val) };
        s.u = u; s.v = v;

        // UNCHECKED INDEXING
        unsafe { rs.get_unchecked(s.v as usize) .reserve(i as
    DefInt); }
        true
    };

    let commit = |i: usize, s: &mut SFState| -> bool {
        unsafe {
            if rs.get_unchecked(s.v as usize).check(i as
  DefInt) {
                uf.link(s.v, s.u);
                true
            } else { false }
        }
    };
    // ... spec_for execution ...
    let res = rs.par_iter().filter_map(|r| /*...*/).collect()
  ;

    // RECYCLE: Return vector to cache
    *rs_cache = Some(rs) ;
    res
}
```

Figure 13. Code comparison for Spanning Forest. **Left (Baseline):** Performs a fresh allocation for the reservation array on every call (red) and uses checked indexing/unwrapping inside the hot loop. **Right (ParEVO):** Implements a memory recycling mechanism via `rs_cache` (green) to reuse the large reservation vector across calls. It also employs `get_unchecked` and `AtomicUnionFind` to eliminate bounds checking and pointer dereference overheads.

RPB (Unsafe Baseline)

```rust
fn process_node(val: ValType, graph: &Graph, data: &
    SharedData,
    pq: &MultiQueue<ValType> // Concurrent Queue Overhead
) {
    let (dist, src) = (val.0, val.1);
    if data.shortest_distance[src].load(Ordering::Relaxed) <
     dist { return; }

    let new_distance = dist + 1;
    for i in graph.nodes[src]..graph.nodes[src + 1] {
        let target = graph.edges[i].target;
        let mut old_distance = data.shortest_distance[target
    ].load(Ordering::Relaxed);

        // HOT LOOP: High Contention Point
        while new_distance < old_distance {
            // HEAVY SYNC: Compare-And-Swap loop
            match data.shortest_distance[target]
    .compare_exchange_weak(
                old_distance, new_distance,
                Ordering::SeqCst, // Strong Ordering
                Ordering::Relaxed,
            ) {
                Ok(_) => {
                    // QUEUE PUSH: Locking overhead
                    pq.push(ValType(new_distance, target));
                    break;
                },
                Err(x) => old_distance = x, // Retry on
    failure
            }
        }
    }
}
```

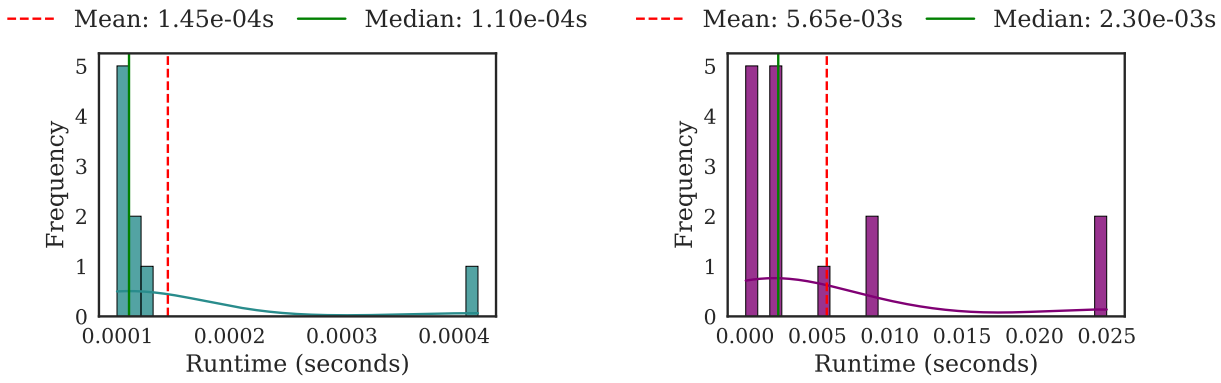
ParEVO (Optimized)

```rust
impl<'a, Fa, Cond> EdgeMap<'a, Fa, Cond> {
    pub fn apply(&self, frontier: VertexSubset) ->
     VertexSubset {
        let n = self.g_out.num_nodes();
        let m = self.g_out.num_edges();

        // HEURISTIC: Check frontier density
        if frontier.is_sparse {
            let l = frontier.sparse.len();
            // Calculate exact workload
            let out_degree = delayed::reduce_map(&fview, |v|
    degree(self.g_out, v));

            // THRESHOLD: Switch based on Edge Count vs
    Vertices
            if l + out_degree > m / 20 {
                // PULL PHASE (Dense Optimization)
                // Scans unvisited nodes to find ANY parent (
    Early Exit)
                // Uses Transpose Graph (g_in) significantly
    reducing checks
                let next_dense =
    edge_map_dense(self.g_in, ...);
                VertexSubset::from_dense(next_dense)
            } else {
                // PUSH PHASE (Sparse Standard)
                // Traditional BFS only for small frontiers
                let next_sparse =
    edge_map_sparse(self.g_out, ...);
                VertexSubset::from_sparse(next_sparse)
            }
        } else {
            // ... Dense -> Dense or Dense -> Sparse logic
    ...
        }
    }
}
```

Figure 14. Code comparison for BFS. **Left (Baseline):** Uses a standard asynchronous approach where every edge relaxation requires a CAS loop and a queue push (red), leading to high contention on scale-free graphs. **Right (ParEVO):** Implements Direction-Optimizing BFS (Ligra-style). It dynamically switches between "Push" (Sparse) and "Pull" (Dense) modes based on the frontier density (green), drastically reducing edge checks during the heavy middle levels of the traversal.



(a) Base Model Runtime Distribution

(b) Fine-tuned Runtime Distribution

Figure 15. Runtime Histograms for Graph Shortest Path. The fine-tuned model (b) exhibits tighter variance (reliability) but a higher median runtime due to overhead from safety-focused primitives.

Baseline: Gemini-2.5-Pro

Runtime: 0.0000775394

```
{
  if (source == dest) { return 0; }

  // ATOMIC DISTANCE ARRAY
  // Allows safe concurrent updates without locks
  auto distances = parlay::sequence<std::atomic<int>>(N);
  parlay::parallel_for(0, N, [&](size_t i) {
    distances[i].store(-1);
  });
  distances[source].store(0);

  parlay::sequence<int> frontier(1, source);
  int current_dist = 0;

  while (!frontier.empty()) {
    if (distances[dest].load() != -1) return distances[dest].
      load();
    current_dist++;

    // Parallel neighbor generation
    auto all_neighbors = parlay::flatten(parlay::map(frontier
      , [&](int u) {
      auto indices = parlay::iota(N);
      return parlay::filter(indices, [&](int v) {
        return A[static_cast<size_t>(u) * N + v] == 1;
      });
    }));

    auto next_frontier = parlay::filter(all_neighbors, [&](
      int v) {
      int expected = -1;
      // ATOMIC CAS: Thread-safe claim of visitation
      return
      distances[v].compare_exchange_strong(expected, current_dist);

    });

    frontier = std::move(next_frontier);
  }
  return -1;
}
```

Finetuned: Gemini-2.5-Parlay

Runtime: 0.0020898816

```
{
  // HEAVY INITIALIZATION (O(N^2) Copy)
  // Reconstructs adjacency matrix explicitly before starting
  auto adj = parlay::tabulate(N, [&](size_t i) {
    return parlay::tabulate(N, [&](size_t j) {
      return A[i * N + j];
    });
  });

  if (source == dest) return 0;
  parlay::sequence<int> dist(N, -1);
  parlay::sequence<int> q; // Standard non-concurrent
      structure
  q.push_back(source);
  dist[source] = 0;

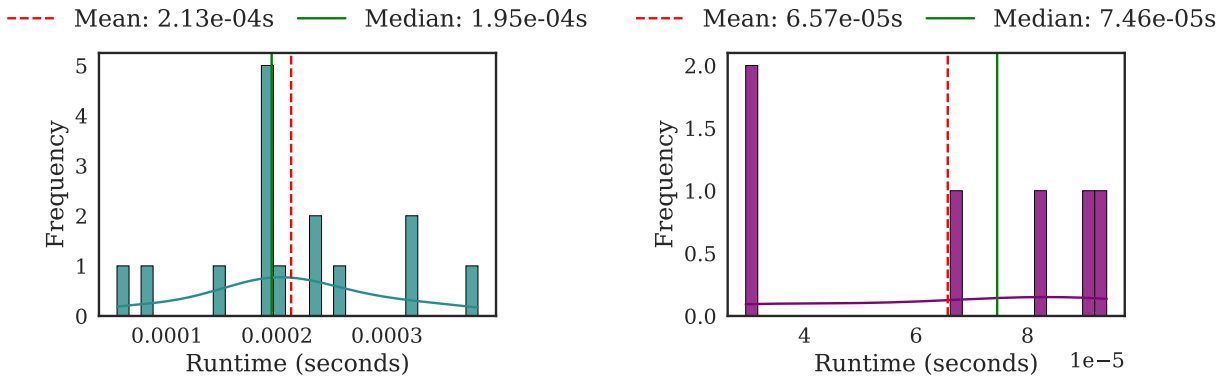
  size_t head = 0;
  while (head < q.size()) {
    size_t current_size = q.size();
    // ... neighbor gathering ...
    auto new_frontiers = parlay::flatten(/*...*/);

    if (new_frontiers.empty()) { head = current_size;
      continue; }

    // SORT/UNIQUE OVERHEAD
    auto unique_frontiers = parlay::unique(new_frontiers);

    // SEQUENTIAL BOTTLENECK
    // Updates and pushes must happen serially here
    for (int v : unique_frontiers) {
      dist[v] = dist[q[head]] + 1;
      if (v == dest) return dist[v];
      q.push_back(v);
    }
    head = current_size;
  }
  return -1;
}
```

Figure 16. Code comparison for Shortest Path (Problem 19 in ParEval). **Left (Baseline):** Effectively uses `std::atomic` and Compare-and-Swap (CAS) to manage visitation state in parallel, resulting in a significantly faster runtime. **Right (Finetuned):** Chooses a high-overhead initialization step (copying the adjacency matrix via `tabulate`) and falls back to sequential logic for the queue update loop (red), causing $O(N^2)$ startup cost and serialization bottlenecks. Nonetheless, it shows heavier abstraction usage.



(a) Base Model Runtime Distribution

(b) Fine-tuned Runtime Distribution

Figure 17. Runtime Histograms for ParEval Problem 34 (Scan). The fine-tuned model (b) exhibits tighter variance and highly predictable performance compared to the wider distribution of the base model (a).

ParEval Prompting Specification

System Instruction:
Fixed instruction prepended to all queries.

```
You are a **helpful** coding assistant.
You are helping a programmer write a C++ function. Write the body of the function and put it in a markdown
    code block.
**Requirements**:
- **DO NOT WRITE ANY COMMENTS OR EXPLANATIONS** in the code!!! Generate **PURE** code!!!
- Before you return the code, make sure to **remove any comments or explanations** that you may have added.
```

C++ User Template:

```
Complete the C++ function {function_name}. Only write the body of the function {function_name}.
```cpp
{prompt}
```
```

Rust User Template:

```
Complete the Rust function {function_name}. Only write the body of the function {function_name}.
```Rust
{prompt}
```
```

Figure 18. The prompting strategy adopted from the ParEval paper (Nichols et al., 2024). The templates include specific placeholders (`function_name`, `prompt`) populated dynamically during evaluation.

ParEval Extension Examples

C++ Prompt (ParlayLib):

```cpp
/* Compute the discrete fourier transform of x. Store the result in output.
   Use ParlayLib to compute in parallel.
   Example:
     input: [1, 4, 9, 16]
     output: [30+0i, -8-12i, -10-0i, -8+12i]
*/
void dft(parlay::sequence<double> const& x,
         parlay::sequence<std::complex<double>> &output) {
```

Rust Prompt (Rayon):

```rust
/* Compute the discrete fourier transform of x. Store the result in output.
   Use Rust Rayon to compute in parallel.
   Example:
     input: [1, 4, 9, 16]
     output: [30+0i, -8-12i, -10-0i, -8+12i]
*/
pub fn dft(x: &[f64], output: &mut [num_complex::Complex<f64>]) {
```

Figure 19. Representative examples of our manual extensions to the ParEval dataset. The prompts are tailored to enforce specific parallel backends while maintaining identical input/output specifications.

PBBSBench Strategy: Concise Prompt

System Instruction:

```
You are an expert C++ programmer with extensive experience in parallel programming.
Write a parallel {} procedure in C++ that is correct and is the fastest parallel {} program you can generate
    .
Return the code between `// --- Start of file:` and `// --- End of file:` markers.
```

User Input (Example: Maximal Independent Set):

```
Returns a maximal independent set for an undirected graph. Use ParlayLib to compute in parallel.

#include "common/graph.h"

using vertexId = uint;
using edgeId = uint;
using Graph = graph<vertexId,edgeId>;

parlay::sequence<char> maximalIndependentSet(Graph const &G);
```

Figure 20. An example of the *Concise* prompt formulation for the PBBSBench `maximalIndependentSet` task. The model is provided with the function signature and a request to use ParlayLib, but implementation details of the `Graph` structure are omitted.

RPB Prompt Structure

Part 1: Context (Excerpt of ParlayLib-Rust Primitives)

The prompt begins by providing the full suite of helper functions (truncated here for brevity).

```
// Here are the primitives you may use
// ... [Full list of primitives omitted] ...

/* -------------------- Flatten -------------------- */
pub fn flatten<T>(arr: &[&Vec<T>], dest: &mut Vec<T>)
where T: Copy + Send + Sync + Default {
    // ... implementation details ...
}

// ... [Additional primitives like scan, reduce, etc.] ...
```

Part 2: Task Definition & Signature

The specific algorithm request follows the context.

```
// Given an undirected graph, return a maximal independent set (MIS).
// The input graph can be in any format.
// The code cannot reorder the graph for locality.
// The output must be a sequence of vertices in the MIS (order irrelevant).

#[cfg(feature = "AW_safe")]
use std::sync::atomic::{AtomicU8, Ordering::Relaxed};
use rayon::prelude::*;
use pbbs::common::graph::Graph;

#[path="../../common/spec_for.rs"] mod spec_for;
use spec_for::StatefulSpecFor;

#[derive(Clone)]
struct MISState {
    flag: u8,
}

pub fn maximal_independent_set(g: &Graph) -> Vec<u8> {
    // LLM_OUTPUT_HERE
}
```

Figure 21. An example of the RPB prompting template. We inject the full set of parallel primitive definitions (represented by the `flatten` excerpt in Part 1) prior to the specific task instructions (Part 2) to ground the model in the available Rust-ParlayLib equivalence layer.