

ThreadWeaver: Adaptive Threading for Efficient Parallel Reasoning in Language Models

Long Lian^{1,2,*}, Sida Wang¹, Felix Juefei-Xu¹, Tsu-Jui Fu¹, Xiuyu Li², Adam Yala^{2,3}, Trevor Darrell², Alane Suhr², Yuandong Tian¹, Xi Victoria Lin^{1,*}

¹Meta Superintelligence Labs (MSL), ²UC Berkeley, ³UCSF

*Work done at Meta

Scaling inference-time computation has enabled Large Language Models (LLMs) to achieve strong reasoning performance, but inherently sequential decoding leads to substantial latency, especially on complex tasks. Recent work on adaptive parallel reasoning aims to improve inference efficiency by decomposing the problem-solving process into concurrent reasoning threads when beneficial. However, existing methods on realistic tasks are either limited to supervised behavior cloning or exhibit significant accuracy drops compared to widely-used sequential long chain-of-thought (CoT) baselines. Moreover, many require customized inference engines, complicating deployment. We introduce ThreadWeaver, a framework for adaptive parallel reasoning that achieves accuracy on par with popular sequential reasoning models of comparable size while significantly reducing inference latency. ThreadWeaver's performance stems from three key innovations: **1)** a two-stage parallel trajectory generator that produces large-scale, high-quality CoT data with parallel annotations for supervised fine-tuning; **2)** a trie-based training-inference co-design that enables parallel reasoning on any off-the-shelf autoregressive inference engine without modifying position embeddings or KV caches; and **3)** a parallelization-aware reinforcement learning framework that trains the model to balance accuracy with effective parallelization. Across six challenging mathematical reasoning benchmarks, ThreadWeaver trained atop Qwen3-8B achieves accuracy comparable to cutting-edge sequential reasoning models (71.9% on average and 79.9% on AIME24) while delivering up to 1.53× average speedup in token latency, establishing a new Pareto frontier between accuracy and efficiency.

Date: November 24, 2025

Correspondence: Long Lian at longlian@berkeley.edu



1 Introduction

Large Language Models (LLMs) have advanced automated problem solving across domains demanding complex reasoning, including mathematics, programming and scientific discovery. This process is driven by scaling inference-time computation to produce longer and more detailed chain of thought (CoT) that iteratively navigates the problem space and revises earlier steps (Wei et al., 2022; Zelikman et al., 2022a; Yao et al., 2023). As task difficulty increases, the time required to reach a correct solution rises dramatically, resulting in tens of hours of continuous computation for a single task (Anthropic, 2025). On the other hand, the sequential nature of autoregressive LLMs imposes severe inference latency. Moreover, because generating each token requires conditioning on all previous tokens, *latency cannot be easily reduced by adding more compute resources*.

To address this limitation, parallel reasoning has emerged as a popular technique for enhancing the performance of LLMs, offering an additional test-time scaling dimension beyond extending sequence length. Methods such as best-of-N and self-consistency (Wang et al., 2023) execute multiple independent reasoning threads and aggregate their outputs, improving answer quality but incurring substantial redundant computation. Test-time search approaches such as tree-of-thought (Yao et al., 2023) guide exploration through hand-crafted structures tuned to the underlying problem, but the manually designed search heuristics are limited by scalability and generality. More recently, several works have begun to explore *adaptive parallel reasoning*, where LLMs are trained to decide when to branch reasoning into parallel threads and when to continue sequentially (Pan et al., 2025; Yang et al., 2025b; Jin et al., 2025; Zheng et al., 2025). These models are trained to emit special tokens

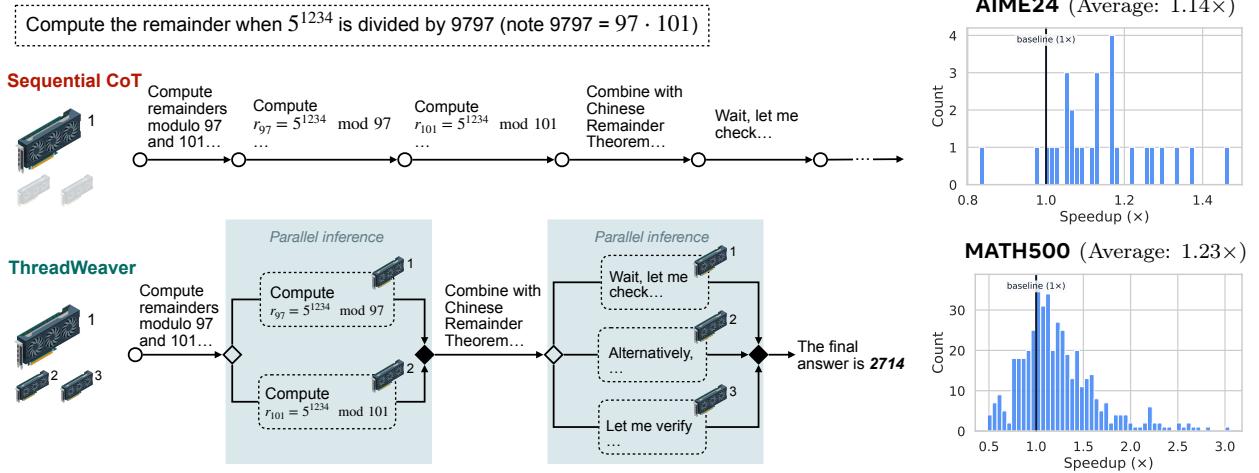


Figure 1 Left: Sequential reasoning solves the problem step by step iteratively, so its reasoning latency grows proportionally to the length of the reasoning chain and cannot be reduced by allocating more compute. ThreadWeaver instead creates concurrent reasoning threads adaptively that tackle different parts of the solution through *spawn* and *join* operations, effectively shortening the critical path when additional compute is available. **Right:** Per-problem speedup histograms on AIME24 and MATH500 for ThreadWeaver showing up to 3 \times per-problem speedup in token latency. This acceleration is achieved without loss in accuracy.

that, during inference, trigger *spawn* and *join* operations to structure the reasoning process via multiple, possibly parallel inference calls, which mimics multi-threaded execution. Such adaptive approaches hold the promise of allocating test-time computation more effectively across reasoning stages, enabling the model to self-determine when parallelization is beneficial for the task at hand.

However, despite its potential, **adaptive parallelization has not yet been demonstrated on complex real-world problems with accuracy matching that of strong sequential reasoning models** (Yang et al., 2025a). We identify three key challenges that limit the effectiveness and scalability of current adaptive parallel reasoning approaches.

Challenge 1: High-quality parallel reasoning trajectories for real-world problems are hard to obtain for training. While human annotation demands high domain expertise and costs, state-of-the-art LLMs (OpenAI, 2025; Team, 2025a) struggle to generate high-quality parallel trajectories when prompted, leaving prior datasets small and narrow in scope.

Challenge 2: Most existing approaches depend on customized inference engines. Prior approaches modify the position embeddings, key-value (KV) caches, or attention masks of the LLM to accommodate thread branching and joining (Jin et al., 2025; Zheng et al., 2025; Yang et al., 2025b), making them difficult to be deployed on standard inference frameworks (Kwon et al., 2023; Zheng et al., 2024), thereby slowing down scalability and adoption.

Challenge 3: Reinforcement learning (RL) for parallel reasoning remains underexplored and difficult to scale. RL over parallel reasoning trajectories introduces additional modeling and system challenges, including advantage calculation across branches and maintaining consistency between train-time rollouts and test-time execution.

To address these challenges, we propose **ThreadWeaver**, a framework that brings *RL-induced adaptive parallel reasoning* to strong reasoning LLMs such as Qwen3 (Yang et al., 2025a) without modifying the underlying inference engine. We target the setting in which abundant compute is available to reduce reasoning latency: most user queries are relatively simple and can be served efficiently via high-throughput sequential decoding, while a smaller fraction of hard problems, often the ones with the highest economic value, justify allocating additional parallel compute to reduce wall-clock time cost. In this regime, conventional data parallelism saturates once inference batch size drops to one, and further latency reduction requires parallelism *within* a single reasoning trajectory. ThreadWeaver is designed specifically to operate effectively under this paradigm.

ThreadWeaver's efficiency improvement stems from three key innovations:

1. **A Two-Stage Parallel Trajectory Generator**, which efficiently converts high-quality sequential reasoning traces into parallel reasoning traces with LLM-annotated control tags and thread summaries, enabling low-cost data generation as a cold-start for the subsequent RL stage ([Section 4](#)).
2. **Trie-Based Training and Inference Co-Design**, which enables parallel reasoning on standard autoregressive inference engines such as vLLM ([Kwon et al., 2023](#)) and SGLang ([Zheng et al., 2024](#)) without any modification to the underlying LLM inference structure, such as position embeddings or KV caches ([Section 3](#)).
3. **Parallelization-Aware GRPO (P-GRPO)**, which jointly optimizes accuracy and latency reduction through mathematically justified thread-wise advantage broadcast ([Section 5.1](#)) and a parallelization-aware reward design ([Section 5.3](#)), effectively inducing adaptive parallelization while maintaining strong reasoning performance.

ThreadWeaver achieves accuracy comparable to equally sized cutting-edge sequential reasoning models (e.g., 79.9% accuracy for ThreadWeaver vs 78.3% for Qwen3-8B ([Yang et al., 2025a](#)) on AIME24, and 71.9% vs 72.2% on average across six widely used math benchmarks) while delivering substantial reductions in token latency: $1.14\times$ speedup on AIME24 ([Mathematical Association of America, 2024](#)), $1.16\times$ speedup on AMC23 ([Mathematical Association of America, 2023](#)), $1.23\times$ speedup on MATH500 ([Hendrycks et al., 2021](#)), $1.21\times$ on OlympiadBench ([He et al., 2024](#)), and $1.53\times$ on Minerva Math ([Lewkowycz et al., 2022](#)). These results establish a new Pareto frontier between accuracy and efficiency. Furthermore, ThreadWeaver offers the foundation for scalable parallel reasoning that remains fully compatible with standard inference engines, opening a promising direction for enabling LLMs to tackle increasingly complex problems within feasible inference time budget.

2 Preliminaries

2.1 Adaptive Parallel Reasoning

Many reasoning problems naturally contain parallelizable structure, such as independent sub-cases, alternative solution paths, or self-reflection on partial results that can be handled along with exploration. Adaptive parallel reasoning (APR) ([Pan et al., 2025](#)) is a reasoning paradigm in which a language model dynamically decides when to exploit such structure within a single inference trajectory.

Formally, let π_θ be a policy over an extended action space that includes text token emissions and two control actions, `spawn` and `join`. Executing `spawn` at time t creates $m \geq 1$ child threads $\{\tau^{(j)}\}_{j=1}^m$ that evolve independently with no cross-thread reads under π_θ until a `join`. At `join`, a designated reducer aggregates the child outputs (e.g., via concatenation or a learned summarization pattern) and returns the resulting information back to the parent thread, which then continues decoding sequentially.

APR ([Pan et al., 2025](#); [Yang et al., 2025b](#); [Zheng et al., 2025](#)) differs from *inter-trajectory* parallelism such as self-consistency and majority vote ([Wang et al., 2023](#)), where multiple complete chains are sampled independently and combined only at the end. In APR, branching and joining occur *inside* a single trajectory with explicit coordination across threads. This allows the model to parallelize decomposable subproblems while maintaining a single coherent solution path. Our framework, ThreadWeaver, follows this paradigm.

2.2 Test-time Speed-Accuracy Pareto Frontier

Test-time scaling methods trade additional inference compute for higher accuracy. Prior evaluations typically emphasize accuracy alone, e.g., by generating longer chains of thought or sampling more trajectories for majority vote, while overlooking the accompanying latency costs.

We instead adopt a **speed-accuracy Pareto view**: given a difficult problem and additional compute, we ask whether a method can *reduce the end-to-end time from receiving a user prompt to producing the final answer*, rather than merely increasing accuracy at the cost of slower inference. In our setting, latency is proxied by **token latency**, defined as the number of tokens on the longest thread, or critical path, of a parallel inference trajectory ([Section 6.1](#)). For sequential models, this reduces to the total number of generated tokens. This

metric correlates well with wall-clock time when parallelism is utilized at inference. Unlike Multiverse (Yang et al., 2025b) and Parallel-R1 (Zheng et al., 2025), which focus on measuring latency per token and the degree of parallelism within a parallel trajectory, we emphasize the speedup to answer generation by comparing the token latency of a parallel model with a sequential model with comparable reasoning capability under the same prompts.

Viewing existing approaches under this measurement reveals sharp differences. Self-consistency (Wang et al., 2023) and other ensemble approaches improve robustness by sampling multiple full solutions in parallel, yet the critical-path latency remains that of an entire trajectory. Moreover, because trajectories are sampled independently without coordination, they often replicate large portions of the same reasoning process, leading to substantial redundant computation. In contrast, APR introduces structured *intra-run* branching: the model decomposes a problem into multiple threads that execute concurrently and then rejoin. When this decomposition aligns with the underlying problem structure, the longest thread can be significantly shorter than a monolithic long chain-of-thought, yielding strictly lower token latency.

3 Model

3.1 Parallel Trajectory Format

Toy example of our *inference-time* trajectory format.

Prompt: What is the distance, in units, between the points $(2, -6)$ and $(-4, 3)$? Express your answer in simplest radical form. Think step by step and put your answer in `\boxed{}`.

Response:

`<think>`

We will use the distance formula $d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$.

`<Parallel>`

`<Outlines>`

`<Outline>1: Compute the squared horizontal difference $(\Delta x)^2$.</Outline>`

`<Outline>2: Compute the squared vertical difference $(\Delta y)^2$.</Outline>`

`</Outlines>`

`<Thread>1: $\Delta x = 2 - (-4) = 6$, so $(\Delta x)^2 = 6^2 = 36$.</Thread>`

`<Thread>2: $\Delta y = -6 - 3 = -9$, so $(\Delta y)^2 = (-9)^2 = 81$.</Thread>`

`</Parallel>`

Sum the results: $36 + 81 = 117$.

Distance $d = \sqrt{117} = \sqrt{9 \cdot 13} = 3\sqrt{13}$.

Therefore, the answer is `\boxed{3\sqrt{13}}`.

`</think>`

The distance between $(2, -6)$ and $(-4, 3)$ is `\boxed{3\sqrt{13}}`.

Figure 2 Format for parallelized reasoning trajectories. `<Parallel>` encloses a fork-join block with `<Outlines>` and multiple `<Thread>` sections; *content inside each `<Thread>` is intended to be generated concurrently* by the runtime, while all other spans are decoded sequentially. This toy snippet is for illustration only since the actual trajectories can be more than 10k tokens.

We extend standard autoregressive (AR) generation with lightweight control tokens arranged in a fork-join pattern to realize adaptive parallel reasoning, following Multiverse (Yang et al., 2025b). Unlike Multiverse, our framework restricts trajectories to a single level of parallelization in which all branches rejoin the main thread. In practice, we found that non-nested structures cover the vast majority of use cases while avoiding the engineering overhead and data-quality challenges introduced by nested parallelization branches. The

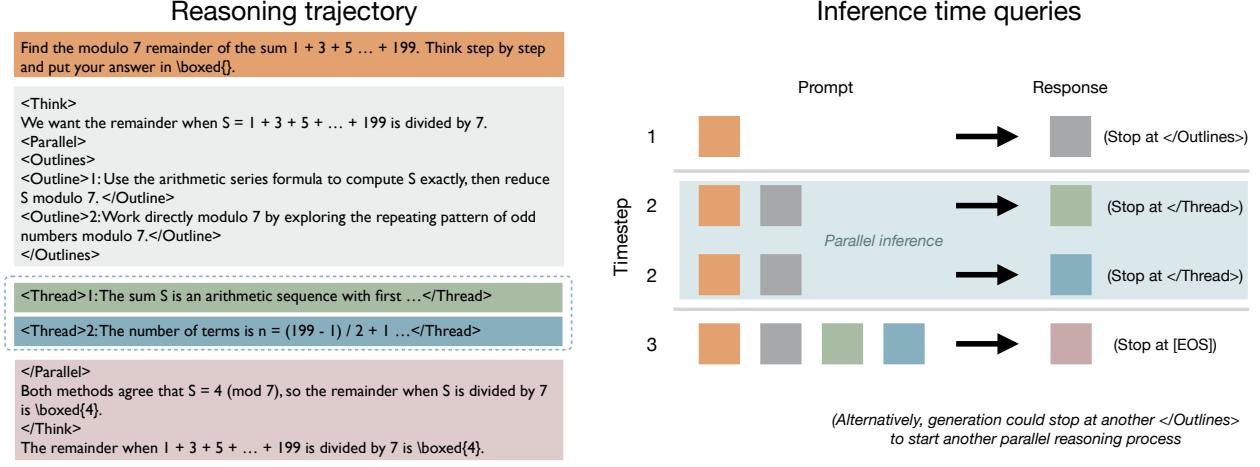


Figure 3 Inference-time request sequence for a parallel reasoning trajectory. For a given user prompt, Timestep 1 decodes the prefix and the `<Outlines>` block sequentially up to `</Outlines>`. Timestep 2 launches one completion request per outline in parallel, each seeded with its corresponding `<Thread> i`: prefix and stopped at `</Thread>`. Timestep 3 resumes sequential decoding over the joined context until [EOS] or the next `</Outlines>`.

underlying techniques proposed in our work naturally generalize to nested branches by permitting parallel blocks within individual threads. Below we describe our parallel trajectory format, with an illustrative example in Figure 2:

- `<think>` marks the start of the reasoning trajectory, which may contain sequential segments and zero or more parallel blocks.
- `<Parallel> ... </Parallel>`: defines a parallel block.
- `<Outlines> ... </Outlines>`: consists of numbered `<Outline>` entries which declare independent sub-tasks.
- `<Thread> i`: execution trajectory of the i -th sub-task. **Threads are generated independently and must not reference other threads.** At inference, each thread is generated in parallel and stopped at `</Thread>`.
- Closing `</think>` marks the end of the reasoning trajectory.

This structured trajectory is generated by a runtime orchestrator described below. The orchestrator spawns parallel generation for each `<Thread>` while decoding all other segments autoregressively, allowing the full trajectory to be generated without any modifications to the underlying inference engine.

3.2 Parallel Inference

As shown in Figure 3, our parallel structure introduces `<Parallel>` blocks that contain an `<Outlines>` section followed by multiple `<Thread>`'s. Importantly, *only the thread contents are generated in parallel*, while all remaining parts of the trajectory are generated via standard autoregressive (AR) decoding. This design allows parallel inference to be implemented on top of any model exposed through a request-completion interface, such as an LLM inference server with API backed by an autoregressive inference engine like SGLang (Zheng et al., 2024) or vLLM (Kwon et al., 2023), without any modifications to the underlying system.

State-machine view. We implement inference as a minimal state machine operating on request-response pairs with a text-completion API: in each step, the orchestrator sends the context (the request) in a single API call and receives the model's completion up to a registered stop token (the response). The orchestrator state machine includes the following phases, starting with a sequential phase with the user prompt wrapped in a chat template defined by the model as the initial context:

1. **Sequential phase.** Decode sequentially until `</Outlines>` is generated, which is registered as a stop token for the sequential phase.
2. **Parse outlines.** Extract the numbered `<Outline>` entries from the preceding `<Outlines>` block.
3. **Parallel phase.** For each `<Outline>` entry i , issue a completion request with `<Thread> i:` appended to the previous context that includes all tokens that the LLM generated previously, including the `<Outlines>` block. Generate completions independently until `</Thread>`, which is registered as a stop token for independent generation in the parallel phase. Threads do not attend to each other as they are treated as different requests by the inference engine.
4. **Join.** Concatenate the context (prompt, outlines, and all thread texts) and append `</Parallel>` to form the context for the next round of text completion.¹
5. **Continue.** Proceed sequentially until the next `</Outlines>` token or the `[EOS]` token.

Because we introduce no changes to the LLM architecture or context representation, our method inherits existing serving optimizations (e.g., paged attention, prefix caching) and runs out of the box on common LLM inference stacks that expose a text-completion API. Note that common inference engines such as vLLM and SGLang support prefix caching, which prevents regenerating the KV cache (i.e., rerunning prefilling) for context tokens if they share prefix with a prior request. When the host API supports taking and returning token IDs, all steps above can operate directly on token IDs without text decoding for additional efficiency. The main potential sources of overhead are repeated API calls, which can increase queueing and scheduling costs especially in trajectories with many short branches, and needing to prefill for threads beyond the first one, which can be expensive when later branches are much longer. In our experiments, these factors did not become bottlenecks, as reflected in the wall-clock speedups reported in [Section 6.3.3](#), in part because we use an efficient vLLM-based implementation with low scheduling overhead and because prefilling is significantly cheaper than decoding.

Hybrid operation. The inference engine design enables our model to function as a hybrid reasoner with both parallel and sequential reasoning capabilities at inference time. Operators may choose between 1) **parallel mode**, which executes the state machine described above, and 2) **autoregressive mode**, which bypasses the state machine and treats all control tokens as ordinary text, which we also handle in our training data curation pipeline. To our knowledge, ours is the first approach that maintains full compatibility with auto-regressive decoding while still enabling *parallel* reasoning at inference time. The user may choose parallel mode when batch sizes are small and latency is critical, and autoregressive mode when batch sizes are large and throughput is the priority.

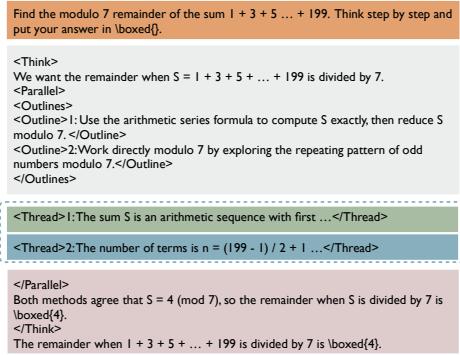
3.3 Efficient Training with Trie-based Sequence Merging

To fine-tune an existing LLM to output the parallel reasoning structures described above, we align training targets with the inference state machine through trie construction and traversal.

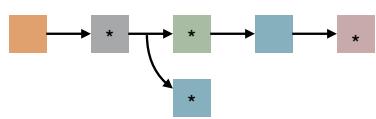
Trie construction and loss computation. As shown in [Figure 4](#), using regex-based parsing, we first extract all the `(context, completion)` units that will be issued during inference. Specifically, each context corresponds to a fragment issued from the client side, and each completion corresponds to the LLM generated span between structural tokens (e.g., from `<think>` to `</Outlines>`). The pair of spans is concatenated into a token sequence. All token sequences retain their markup tokens (e.g., `<Parallel>`, `<Outline>`, `<Thread>`) because these tokens drive the structural parsing later on. We insert all units into a token-level prefix tree whose root is the shared prompt, with each node corresponding to a span (either context or completion) in the `(context, completion)` unit. Branches capture divergent continuations (e.g., different thread texts). Nodes in the trie inherit ancestor context but are isolated from siblings, with the whole trie encoding all trajectories without duplication. Finally, we use depth-first traversal to linearize the trie into a single training sequence.

¹While we expose all the reasoning in subthreads to the main thread, further optimizations are possible to only return necessary information for subsequent computation, reducing the context usage for reasoning.

Step 1: Extracting all <context, completion> pairs from a training example



Step 2: Prefix-Tree Building



Step 3: Tree Flattening to Single Sequence

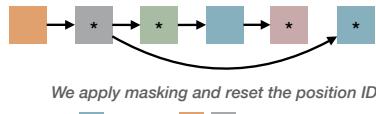


Figure 4 Our training sequence formatting consists of three steps: 1) extract context and completion segments that the inference state machine will encounter in order to produce this trajectory; 2) insert these segments into a token-level trie (prefix-tree); 3) traverse the trie to produce a flat sequence with an ancestor-only attention mask and standard incremental positions counting from the root node. * denotes text spans where loss is applied. Although we only show one branching in this example, ThreadWeaver supports branching and joining multiple times in a trajectory.

We build an *ancestor-only* attention mask: token i may attend to token j if and only if j is an ancestor of i in the trie, which prevents cross-thread leakage while preserving shared prefixes. This flattened sequence includes the merged trajectory and all branch-specific continuations. The trajectories and continuations are interleaved, with position identifiers assigned accordingly, so that each example from Step 1 appears as a contiguous subsequence.² Loss is applied only to tokens in the completion span and is not applied to contextual tokens. Since each context-completion pair can be found as a subsequence from the linearization of the trie, training on the linearization would allow the model to generate the desirable completion given each context specified in Section 3.2. Our trie-based sequence merging technique is naturally applicable to trajectories with multiple <Parallel> blocks, with additional implementation details in Section B.

Consistency and scalability. This co-design guarantees that all the tokens in each request-response pair that the orchestrator issues will have exactly the same position embeddings and context as in the trajectory generated by the trie traversal. Training with trajectories constructed in this manner also ensures that disabling parallel blocks recovers ordinary autoregressive reasoning behavior, since the overall trajectory obtained by autoregressive inferencing is always a valid subsequence of the trajectory obtained from the trie. We employ FlexAttention (Dong et al., 2024) for efficient attention masking in training.

4 Parallel Trajectory Curation for Supervised Fine-tuning

With our proposed inference pattern and training paradigm, a key challenge is obtaining suitable training data for supervised fine-tuning, especially for real, competition-level math problems, where high-quality trajectories are difficult to obtain. Our training procedure requires parallel reasoning trajectories in the fork-join format described in Section 3.1, ensuring that SFT targets align with both the runtime orchestrator in Section 3.2 and the trie-aligned training in Section 3.3. To construct these trajectories, we employ a two-stage pipeline: 1) an LLM-driven annotation and light-rewriting phase for SFT cold start, followed by 2) self-training with reward-based filtering to scale data quality and improve performance.

²The change in attention masks and position IDs at training time are solely to allow example packing for reducing re-computation during training. No changes in KV cache handling, attention masks, or position IDs are required at inference time, in contrast to Yang et al. (2025b) which requires changes in the core module of SGLang due to specialized KV cache handling.

4.1 Stage 1: LLM Rewriting

We first extract sequential reasoning trajectories by running inference on Qwen3-8B (Yang et al., 2025a) with a dataset of prompts. We run a five-step annotation pipeline that uses an LLM (GPT-5 in our case) to identify and annotate parallelizable spans on top of existing Qwen3-8B trajectories while leaving most tokens untouched. The design follows two principles: **a)** the LLM should recover structure and parallelism from a given chain-of-thought transcript rather than regenerate the entire solution, and **b)** rewriting should be applied only where needed to improve clarity, enforce independence between branches, and smooth transitions at the boundaries of parallel blocks, preserving the exploration and self-reflection in the reasoning trajectories.

Identification of parallel blocks. In the first step, each raw reasoning trajectory is line-numbered and processed through a sequence of prompts that output a structured summary with explicit parallel groupings, where every step, substep, and proposed parallel block is tied to precise spans based on line numbers. The prompts used in this step for identifying the parallel regions are constructed with reference to the prompts in Multiverse (Yang et al., 2025b), but ours operate directly on line numbers and do not require the LLM to reconstruct the full trajectory during rewriting.

Canonical thread extraction. The line-number based parallel annotations are then converted into our fork-join markup. The script infers the exact boundaries of each reasoning branch and wraps the corresponding contiguous spans inside `<Thread>` containers under the appropriate `<Parallel>` block. This pass enforces contiguity and ordering constraints, so that each thread corresponds to a well-formed slice of the original trajectory that downstream tools can manipulate without re-parsing the raw text.

Rewriting for clarity. For each thread, we then apply a targeted language-editing pass to remove remaining cross-thread dependencies and to smooth local transitions. The LLM receives an “allowed” context window, a “forbidden” parallel context, and the thread under review, and outputs a list of minimal substring replacements of the form (source text, destination text). These replacements delete or rewrite only the phrases that implicitly refer to the forbidden branch, such as anaphora or shorthand references whose definitions appear solely in other threads, while keeping references to the allowed context intact. Unlike prior work, *the model is never asked to regenerate the full chain-of-thought*, which avoids information loss and keeps the processed trajectories tightly aligned with the original Qwen3-8B rollouts.

Adding outlines. Once the threads are sanitized, our pipeline generates an outline for each branch. Conditioned only on the context that precedes a parallel block and the `<Parallel>/<Thread>` structure, the LLM generates concise, path-specific plans that describe the intended reasoning procedure without revealing intermediate calculations or final conclusions. Each outline is aligned with a single thread and is derivable from the shared context alone without the information from other threads, which makes the resulting `<Outline>` blocks suitable as supervision for planning.

Format checks. Finally, a comprehensive filtering pass traverses the annotated trajectories and discards any samples that violate structural or semantic constraints, such as empty or degenerate threads after rewriting. The output of Stage 1 is therefore a collection of trajectories whose fork-join structure is both syntactically well-formed and semantically justified.

Using this pipeline, we generate 53k sequential reasoning trajectories from Qwen3-8B on Polaris-53k (An et al., 2025), a math problem dataset with problems of varying difficulties and verifiable rewards. We then sample 1k correct trajectories and apply the five-step annotation and rewriting procedure described above to obtain 959 high-quality, parallelized trajectories for cold-start training. Unlike prior approaches that require an LLM to regenerate entire long chains of thought, our pipeline operates through localized edits and structural annotation, which reduces cost and preserves the fidelity of the original reasoning traces. Full details of the pipeline, including the prompts we used for annotation in each step, can be found in Appendix E.

4.2 Stage 2: Self-training with Reward-based Filtering

Although effective, LLM-rewritten trajectories face two limitations: **1)** they cannot be scaled easily due to the high cost of querying LLMs, and **2)** they reflect post-hoc rewrites rather than the generative patterns of the target reasoning model, leaving a gap between the data for SFT and the desired behavior of the model that we want to train. Thus, the rewritten trajectories may not align with how the model plans during generation.

Inspired by prior work on self-improvement methods such as filtered behavior cloning and STaR (Zelikman et al., 2022b), we propose to self-train the target model so that the data distribution matches its own parallel rollouts generated ad hoc. We first fine-tune Qwen3-8B on the 959 cold-start items using trie-based sequence merging (Section 3.3). We then run *parallel* inference with the state machine-based orchestrator, as described in Section 3.2, on the full 53k prompts, producing one trajectory per prompt. Each trajectory receives programmatic feedback: *format correctness* (passes the structural validator above) and *answer correctness* (verifier matches the boxed answer). We keep only items that satisfy both, yielding a large number of self-generated, format-faithful trajectories. Running SFT on this dataset stabilizes the generation of parallel structure and aligns supervision with the model’s own generative patterns.

In summary, while Stage 1 provides a strong prior for parallelized reasoning with minimal edits, Stage 2 allows scaling and bridges the train-test gap. Together, these stages produce a high-quality SFT dataset for parallelized reasoning, enabling reliable parallel rollout and training in subsequent RL.

5 Reinforcement Learning with Parallelization-Aware Reward

5.1 Preliminary: GRPO

Group Relative Policy Optimization (GRPO) is a policy-gradient method that computes advantages with intra-prompt normalization (Shao et al., 2024). For a prompt p , let $\mathcal{G}_p = \{r_1, \dots, r_k\}$ be the scalar rewards for k trajectories sampled from p . The group-normalized advantage for the trajectory i is

$$A_{p,i}^{\text{GRPO}} = \frac{r_i - \mu_p}{\sigma_p + \varepsilon}, \quad \mu_p = \frac{1}{k} \sum_{j=1}^k r_j, \quad \sigma_p = \sqrt{\frac{1}{k} \sum_{j=1}^k (r_j - \mu_p)^2}, \quad (1)$$

with a small $\varepsilon > 0$ for numerical stability. This yields advantages that are comparable across prompts, mitigating reward-scale and difficulty variance.

While GRPO can be paired with a PPO-style clipping surrogate (Schulman et al., 2017), in our setting we use fully *on-policy* updates (single rollout, single optimization step), so the importance ratio is effectively 1 and clipping is inactive. We thus optimize the score-function objective

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{p,i,\tau_p^{(i)} \sim \pi_\theta} \left[A_{p,i}^{\text{GRPO}} \sum_{t=1}^{T_{p,i}} \log \pi_\theta(a_t | s_t) \right], \quad (2)$$

whose gradient is obtained from the standard REINFORCE estimator. In practice, we minimize the pseudo-loss

$$\mathcal{L}_{\text{GRPO}}(\theta) = -\frac{1}{\sum_{p \in \mathcal{B}} \sum_{i=1}^k T_{p,i}} \sum_{p \in \mathcal{B}} \sum_{i=1}^k A_{p,i}^{\text{GRPO}} \sum_{t=1}^{T_{p,i}} \log \pi_\theta(a_t | s_t), \quad (3)$$

where $T_{p,i}$ is the number of completion tokens for the trajectory $\tau_p^{(i)}$. This loss has a gradient $-\nabla_\theta \mathcal{J}_{\text{GRPO}}$. Note that we follow DAPO (Yu et al., 2025) and normalize over tokens across sequences in the batch ($\sum_{p \in \mathcal{B}} \sum_{i=1}^k T_{p,i}$).

5.2 P-GRPO: GRPO with Parallel Rollout

State-machine based rollout. We perform rollout using the inference orchestrator described in Section 3.2. Let $\tau_p^{(i)}$ be composed of the ordered set of $\langle \text{context}, \text{completion} \rangle$ units emitted by the state machine,

Algorithm 1 P-GRPO (Parallel GRPO) Training Loop

Require: Post-SFT LLM parameters θ ; training prompt set \mathcal{D} ; number of RL iterations N ; group size k ; parallel rollout mechanism (§3.2); reward modules $\mathcal{R}_{\text{correct}}, \mathcal{R}_{\text{accel}}$

- 1: **for** $j = 0, 1, \dots, N - 1$ **do** ▷ RL training iterations
- 2: Sample a minibatch $\mathcal{B} \subset \mathcal{D}$ of prompts
- 3: **for** each $p \in \mathcal{B}$ **do**
- 4: Roll out k parallel trajectories $\{\tau_p^{(i)}\}_{i=1}^k$ with π_θ
- 5: Compute $r_{p,i}$ for each $\tau_p^{(i)}$ using Equations (5) and (7)
- 6: Compute μ_p and $A_{p,i}^{\text{P-GRPO}}$ via Equations (1) and (9)
- 7: Broadcast $A_{p,i}^{\text{P-GRPO}}$ to all tokens of $\tau_p^{(i)}$
- 8: **end for**
- 9: Form the loss $\mathcal{L}_{\text{P-GRPO}}(\theta)$ in Equation (4)
- 10: Update parameters with a single on-policy gradient step with gradient $\nabla_\theta \mathcal{L}_{\text{P-GRPO}}(\theta)$
- 11: **end for**

$\tau_p^{(i)} = \{u_p^{(i,1)}, u_p^{(i,2)}, \dots, u_p^{(i,M_i)}\}$, where $u_p^{(i,m)} = [\text{cont}_p^{(i,m)} \| \text{comp}_p^{(i,m)}]$ is the concatenation of the context $\text{cont}_p^{(i,m)}$ followed by its corresponding completion $\text{comp}_p^{(i,m)}$. Training with these units ensures the distribution seen during RL matches test-time execution so that there is no train–inference gap.

Advantage calculation. We compute a single scalar reward $r_{p,i}$ for the *entire* trajectory (see Section 5.3) and *broadcast* its group-normalized advantage $A_{p,i}$ to all units in that sample. The loss is defined as

$$\mathcal{L}_{\text{P-GRPO}}(\theta) = -\frac{1}{\sum_{p \in \mathcal{B}} \sum_{i=1}^k T_{p,i}} \sum_{p \in \mathcal{B}} \sum_{i=1}^k A_{p,i} \sum_{m=1}^{M_i} \log \pi_\theta(\text{comp}_p^{(i,m)} \mid \text{cont}_p^{(i,m)}). \quad (4)$$

Equation (4) is implemented with a token-level mask that zeros out the loss on context tokens, so the log-probability term only accumulates contributions from completion tokens.

This thread-wise broadcast is mathematically justified, keeps the implementation simple, and empirically stabilizes training without per-branch credit heuristics. Following the practice in sequential reasoning and multi-turn RL, we do not apply loss on the context tokens of each thread, so each generated token receives loss at most once, even if it later reappears as part of the context for subsequent generation. We provide derivations of P-GRPO and mathematical justification for thread-wise broadcast in Section A.

5.3 Parallelization-Aware Reward

Similar to sequential generation, we set the main rewards to an indicator function from answer correctness. In addition, we softly encourage *acceleration* when parallelization reduces the critical path. For sample τ , the total reward includes the two components added together:

$$r(\tau) = \mathcal{R}_{\text{correct}}(\tau) + \mathcal{R}_{\text{accel}}(\tau), \quad (5)$$

The correctness reward $\mathcal{R}_{\text{correct}}(\tau)$ evaluates the boxed final answer:

$$\mathcal{R}_{\text{correct}}(\tau) = \mathbf{1}\{\text{CORRECT}(\tau)\}. \quad (6)$$

The acceleration reward $\mathcal{R}_{\text{accel}}$ incentivizes the model to generate more efficient reasoning paths by leveraging parallelization:

$$\mathcal{R}_{\text{accel}}(\mathbf{s}) = \mathbf{1}\{\text{CORRECT}(\tau)\} \min(\rho \cdot \eta(\mathbf{s}), \rho_{\text{clip}}) \quad (7)$$

where \mathbf{s} denotes the set of parallel threads in the trajectory, ρ is an acceleration reward scale, and ρ_{clip} is a

clipping threshold.³ Acceleration ratio $\eta(\mathbf{s})$ is computed as:

$$\eta(\mathbf{s}) = 1 - \frac{L_{\text{longest}}}{L_{\text{total}}} \quad (8)$$

where L_{longest} is the sequence length for the longest thread (i.e., token latency), L_{total} is the total number of tokens in the entire trajectory. The acceleration ratio reward is only given when the answer is correct, which is indicated by the correctness term in [Equation \(7\)](#).

Removing normalization with standard deviation for stability. Our reward is the sum of a correctness term and an acceleration term, which makes the relative scale between the two components important. Empirically, we found that using the original GRPO advantage with standard-deviation normalization leads to unstable training: the model quickly optimizes for acceleration, while its correctness degrades.

A key failure mode arises when all rollouts for a given prompt are correct. In that case, the correctness reward $\mathcal{R}_{\text{correct}}$ is identically 1 across the group and is therefore removed by mean-centering. Afterward, dividing by the group standard deviation effectively cancels the intended scaling set by the acceleration factor ρ , so the acceleration term dominates without proper control. To preserve the scale of advantages and maintain a stable trade-off between correctness and acceleration, we remove the standard-deviation normalization and instead use advantages based solely on mean-centering

$$A_{p,i}^{\text{P-GRPO}} = r_{p,i} - \mu_p. \quad (9)$$

This choice is aligned with the modifications proposed by [Liu et al. \(2025\)](#), though our motivation originates from the multi-term reward structure proposed for incentivizing parallel reasoning.

Overall, since the rollout uses our parallel inference scheme specified in [Section 3.2](#), P-GRPO learns parallel behavior that transfers directly to any off-the-shelf AR engine, along with correctness and acceleration improvements. We present the pseudo-code for our proposed P-GRPO in [Algorithm 1](#).

6 Experiments

6.1 Setup

We evaluate ThreadWeaver atop Qwen3-8B ([Yang et al., 2025a](#)), a reasoning model that demonstrates strong performance on mathematical problem solving. ThreadWeaver is first trained through a three-stage pipeline, consisting of supervised fine-tuning, self-training, and reinforcement learning, and performs inference with parallelized reasoning.

Supervised Fine-Tuning. To prepare data for supervised fine-tuning (SFT), we first perform inference on the Polaris 53k ([An et al., 2025](#)) prompt set using Qwen3-8B ([Yang et al., 2025a](#)). This produces 53,291 trajectories, one per prompt. Among these, 36,296 trajectories contain both correct answers and valid reasoning structures (i.e., with correct answer in `\boxed{}` after `</think>`). From this subset, we sample 1,000 high-quality examples for rewriting with GPT-5 ([OpenAI, 2025](#)), obtaining 959 trajectories that preserve both correctness and structural validity. These examples are used for the first stage of SFT, where Qwen3-8B is fine-tuned for 8 epochs to teach it to reason in parallel, following the parallel reasoning format specified in [Figure 2](#) and the trie-based sequence merging in [Section 3.3](#).

Self-training. After SFT, we apply self-training with reward-based filtering using the same 53k prompts. The model generates one trajectory for each prompt, and we select those that are both answer-correct and structurally valid, forming a dataset of 17,491 trajectories. A second round of SFT is then conducted on this dataset to further stabilize and strengthen the model’s ability to generate well-structured parallel reasoning.

³In our experiments we set $\rho = 0.5$ and $\rho_{\text{clip}} = 0.2$. These values are chosen empirically to keep the acceleration bonus a small fraction of the correctness reward so that the model does not sacrifice accuracy for acceleration gains.

Table 1 Comparison between the sequential GRPO baseline and ThreadWeaver on Qwen3-8B across six math benchmarks. ThreadWeaver matches the baseline’s accuracy (71.9% vs. 72.2%) while **reducing mean critical-path length from 15.1k to 13.2k tokens** and achieving **up to 1.53 \times average speedup** and **up to 3.56 \times max speedup** on datasets with highly decomposable structure. Speedups are defined by the ratio between sequential and parallel token latencies and are computed in three ways: 1) averaged over all rollouts (Speedup), 2) averaged only over problems where both models produce correct rollouts (Speedup, correct only), and 3) the maximum speedup among correct rollouts, which serves as an empirical upper bound on achievable acceleration.

Model	AIME24 (Avg@32)	AIME25 (Avg@32)	AMC23 (Avg@8)	MATH500 (Avg@1)	Minerva Math (Avg@4)	OlympiadBench (Avg@8)	Average
Accuracy (%)							
Qwen3-8B + Seq. RL	78.3	61.6	92.6	91.8	43.9	65.0	72.2
Qwen3-8B + ThreadWeaver	79.9	60.5	92.3	91.4	43.7	63.5	71.9
Token Latency (tokens)							
Qwen3-8B + Seq. RL	19.4k	24.6k	13.8k	7.2k	10.6k	15.2k	15.1k
Qwen3-8B + ThreadWeaver	16.9k	24.0k	12.0k	6.4k	7.3k	12.8k	13.2k
Speedup	1.14 \times	1.03 \times	1.16 \times	1.23 \times	1.53 \times	1.21 \times	1.22 \times
Speedup (correct only)	1.14 \times	1.05 \times	1.17 \times	1.23 \times	1.47 \times	1.20 \times	1.21 \times
Max Speedup (correct only)	1.47 \times	1.21 \times	1.67 \times	3.05 \times	3.56 \times	1.92 \times	—

Reinforcement Learning. Finally, we conduct reinforcement learning (RL) with our proposed P-GRPO in Section 5.2. We leverage training prompts from the subset of Polaris 53k dataset proposed in An et al. (2025), which is filtered by Qwen3-4B to include samples that are not fully correct or incorrect in order to accelerate training. To perform a fair comparison with sequential reasoning, we perform reinforcement learning on Qwen3-8B with the same training prompts and run sequential RL using GRPO. Both our model and the sequential RL baseline are conducted with 350 training steps, with training batch size 128 and 8 rollouts per prompt. Note that Qwen3-8B is already a reasoning model and thus no SFT is performed prior to RL in the sequential baseline. In contrast to the multi-stage RL pipeline in An et al. (2025) with different context lengths, we only apply one stage of reinforcement learning with 40k context length for simplicity.

Evaluation. We measure performance of our method on six real-world math benchmarks, including AIME24 (Mathematical Association of America, 2024), AIME25 (Mathematical Association of America, 2025), Minerva Math (Lewkowycz et al., 2022), AMC23 (Mathematical Association of America, 2023), OlympiadBench (He et al., 2024), and MATH500 (Hendrycks et al., 2021). We report accuracy and token latency to assess reasoning correctness and efficiency. Accuracy measures whether the final answer in the last \boxed{} in the reasoning trajectory matches the ground truth in the dataset⁴. Token latency is the number of tokens in the longest thread, or critical path, of the trajectory. This is computed with all sequential and parallel segments into account: for each parallel segment, we compute the number of tokens in the longest thread, and for each sequential segment, we compute the number of tokens. The final number of tokens is the sum of the number of tokens for sequential and parallel blocks. To ensure a fair comparison and isolate the effects of RL and parallel reasoning, we train a sequential RL baseline on Qwen3-8B with the same training prompts as parallel RL training and report accuracy and token latency for both sequential and parallel models.

6.2 Main Results

We report the performance of ThreadWeaver in Table 1, comparing it against a sequential GRPO baseline on Qwen3-8B using the same RL prompts across six math benchmarks. In this setting, both models are trained with identical RL configurations and evaluated with multiple rollouts per problem. For each problem and model, we average the token latency over all rollouts to obtain a per-problem mean token latency. The per-problem speedup for each problem is then computed as the ratio between the mean token latencies of the sequential GRPO baseline and our model. We report the average of the speedup across all the problems

⁴If no \boxed{ } tokens are produced, we treat the example as incomplete and thus incorrect, following Luo et al. (2025); An et al. (2025). This can occur for both sequential and parallel rollouts when the trajectory is too long and does not terminate in a final answer. We do not constrain where the \boxed{ } appears in the trajectory, matching the evaluation protocol used for standard sequential model rollouts.

Table 2 Comparison with other adaptive parallel reasoning approaches on AIME24. ThreadWeaver, built on Qwen3-8B (8B), achieves higher accuracy and stronger self-parallelism than Multiverse (32B) and Parallel-R1 (4B). Parallel-R1 often reasons with a much shorter effective sequence length, which explains its substantially lower accuracy relative to commonly used long-context sequential reasoning models of comparable size. Activation ratio is defined as the percentage of samples with parallelization in the inference trajectories. Results for Multiverse and Parallel-R1 are obtained from Yang et al. (2025b) and Zheng et al. (2025), respectively.

Model	Model Size	Self-Parallelism Speedup	Activation Ratio (%)	AIME24 Accuracy
Multiverse-zero (Yang et al. (2025b), Avg@8)	32B	1.04×	-	52.1%
Multiverse (Yang et al. (2025b), Avg@8)	32B	1.18×	-	53.8%
Parallel-R1-Seen (Zheng et al. (2025), Avg@16)	4B	-	27.3	19.4%
Parallel-R1-Unseen (S1) (Zheng et al. (2025), Avg@16)	4B	-	13.6	18.3%
Parallel-R1-Unseen (S2) (Zheng et al. (2025), Avg@16)	4B	-	63.0	16.3%
ThreadWeaver (ours, Avg@32)	8B	1.25×	85.2	79.9%

in each dataset in the **Speedup** row. We also compute the average speedup restricted to rollouts that yield correct final answers, to compare efficiency under the condition that both our model and the baseline solve the problem correctly, as listed in the **Speedup (correct only)** row. For this metric, if there is no correct rollout for either the sequential baseline or our model on a given problem, that problem is excluded from the speedup computation. The **Max Speedup (correct only)** row reports the maximum per-problem speedup among the rollouts that produce correct final answers. This metric captures the best acceleration the model can achieve when solving the problem correctly and serves as an empirical upper bound of our method. The “Average” column aggregates the six dataset-level values in each row.

On average, ThreadWeaver matches the baseline’s accuracy (71.9% vs. 72.2%) while reducing mean token latency from 15.1k to 13.2k and yielding up to 1.53× speedup across six benchmarks. The per-problem speedup metrics further show that ThreadWeaver is able to achieve a substantial length reduction on the critical path compared to the sequential GRPO baseline. Specifically, ThreadWeaver achieves speedups on all the tasks we benchmark on with minimal impact on accuracy. In particular, accuracy on AIME24 increases slightly (+1.6%), and performance on AMC23, MATH500, and Minerva Math remains on par with the sequential model. These results indicate that adaptive parallelization preserves solution quality while improving inference efficiency on diverse, real-world math tasks.

We observe the highest reduction in token latency on benchmarks with long, decomposable reasoning traces (e.g., AIME24, OlympiadBench), where independent subtasks, including algebraic subcases, casework, and intermediate verifications, can be processed concurrently. We also observed smaller speedup compared to sequential GRPO on AIME25. An inspection of the trajectories shows that our model performs more self-reflection on this dataset, potentially due to less effective or confident reasoning in the parallel reasoning regime, leading to less effective latency reduction. We believe this stems from limited training data coverage. Overall, the consistent latency reduction from ThreadWeaver reflects genuine structural acceleration without sacrificing the depth or rigor of the underlying reasoning.

While the token latency metric measures the length of the critical reasoning path, it is exactly proportional to wall-clock runtime only under an idealized autoregressive engine and does not capture system-level overhead such as scheduling, communication, kernel launches, or imperfect hardware utilization. Therefore, the speedups reported in token latency should be interpreted as an upper-bound to end-to-end runtime improvements. We investigate speedups in wall-clock time in Table 4.

Comparisons to Existing Adaptive Parallel Reasoning Approaches. We compare ThreadWeaver with recently proposed adaptive parallel reasoning methods on AIME24, focusing on Multiverse (Yang et al., 2025b) and Parallel-R1 (Zheng et al., 2025). As summarized in Table 2, ThreadWeaver attains substantially higher accuracy (79.9%) compared to Multiverse (up to 53.8%) and Parallel-R1 variants (up to 19.4%). Furthermore, ThreadWeaver achieves higher *self-parallelism speedup* and *activation ratio*. The former is defined as the ratio between the total number of tokens and the token latency in reasoning trajectory, and the latter is defined as

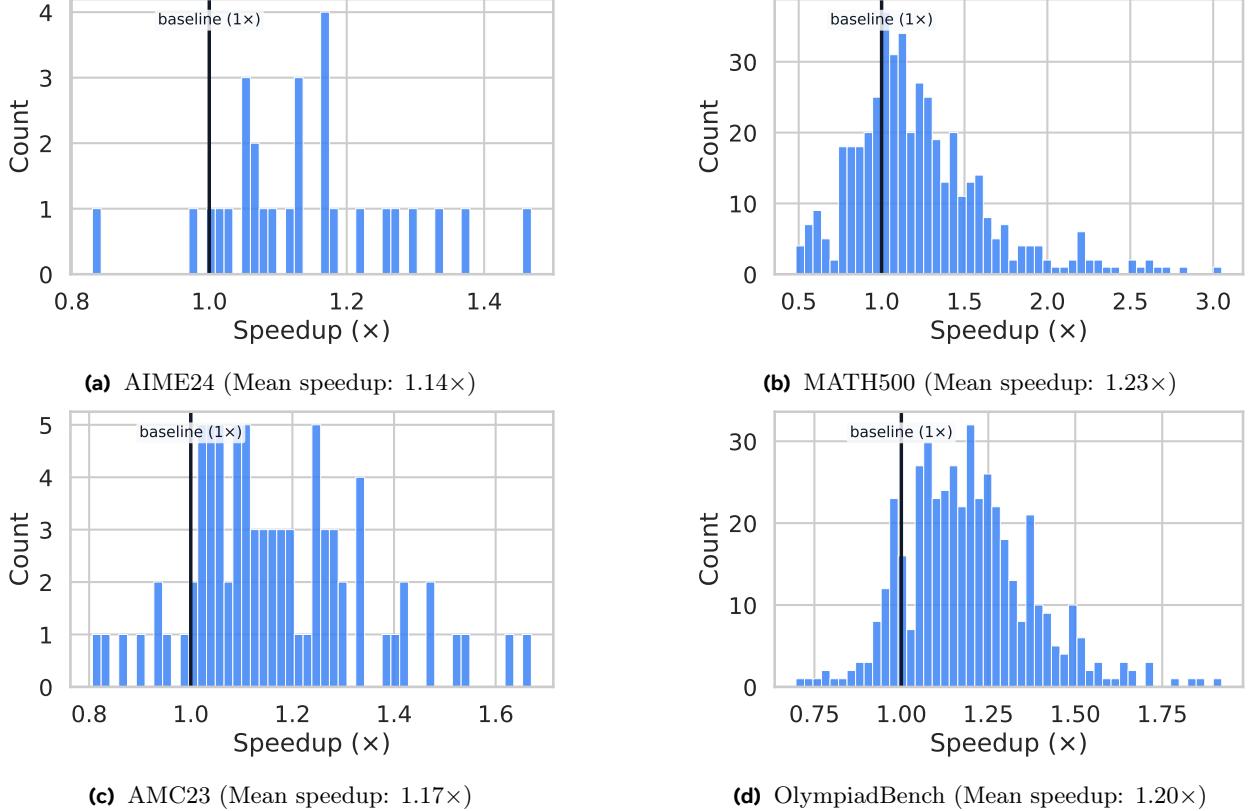


Figure 5 Per-problem speedup distributions on math benchmarks. The speedup is the ratio between the token latency of the sequential reasoning baseline and the token latency of ThreadWeaver. A vertical reference line at $1.0\times$ marks parity with the sequential baseline. ThreadWeaver achieves a significant speedup on most problems.

the ratio between the number of samples with parallelization and the total number of samples in the inference trajectories. **ThreadWeaver achieves both higher accuracy and higher efficiency, establishing a stronger Pareto frontier than prior adaptive parallel reasoning approaches.**

We also observed that ThreadWeaver built on Qwen3-8B outperforms Multiverse, which is built on Qwen2.5-32B (Qwen et al., 2025), both in accuracy and in measured self-parallelism speedup. This indicates that our training recipe effectively induces adaptive parallelization behavior of the model at a much smaller model size. Compared to Parallel-R1, which is also based on a model from the Qwen3 family (Qwen3-4B-Base), ThreadWeaver operates in a long chain-of-thought regime that closely matches the sequence lengths of strong sequential reasoning models in the Qwen3 family. In contrast, Parallel-R1 is trained to produce significantly shorter chains of thought, which limits its test-time scaling ability and accuracy.

6.3 Analysis

6.3.1 Acceleration depends on problem structure

To understand how acceleration varies across problems and datasets, in Figure 5 we show the distribution of per-problem speedups on four math benchmarks. For each problem, we compute the ratio between the token latency of the sequential autoregressive baseline and the token latency of ThreadWeaver, and plot these ratios as histograms. On AIME24/AMC23/OlympiadBench, each ratio is first averaged over 32/8/8 rollouts per prompt, respectively. This is to reduce sampling noise, so each histogram bin reflects differences across questions rather than across samples of the same question. On MATH500, we use one rollout per problem.

All four datasets show substantial mass on the right side of the $1.0\times$ reference line, indicating that ThreadWeaver achieves non-trivial acceleration on most of the questions. In addition, all show a wide spread of

Table 3 Two notions of efficiency. We report 1) *Speedup vs. sequential baseline* = $\frac{L_{\text{baseline longest}}}{L_{\text{ours longest}}}$, which reflects speedup for switching from a sequential reasoning model to a parallel one, and 2) *Self-parallelism speedup* = $\frac{L_{\text{ours total}}}{L_{\text{ours longest}}}$, which reflects internal parallel breadth. While the self-parallelism speedup is informative about a model’s tendency to parallelize, the speedup vs. the sequential baseline is the most direct metric comparing gains from shifting from sequential to parallel reasoning. As in [table 1](#), speedups are computed per-problem and averaged across the problems.

Dataset	ThreadWeaver Longest	Seq. Baseline Longest	Speedup vs. Seq. Baseline	ThreadWeaver Total Tokens	Self-Parallelism Speedup
AIME24	16.9k	19.4k	1.14×	21.1k	1.25 ×
AIME25	24.0k	24.6k	1.03×	27.3k	1.18 ×
AMC23	12.0k	13.8k	1.16×	15.0k	1.25 ×
MATH500	6.4k	7.2k	1.23 ×	7.7k	1.22×
Minerva Math	7.3k	10.6k	1.53 ×	8.4k	1.14×
OlympiadBench	12.8k	15.2k	1.21×	15.5k	1.22 ×

per-problem speedups. Some questions admit large gains when parallelized, while others offer little exploitable structure and occasionally even incur slowdowns relative to the sequential baseline, with speedups as high as around $1.47\times$ for AIME24 and even $3.05\times$ for MATH500. These distributions make clear that acceleration is fundamentally question-dependent: ThreadWeaver can deliver strong critical-path reductions on problems with rich decomposable structure, but naturally falls back toward parity on problems that are intrinsically more sequential.

6.3.2 Critical-path speedup against sequential baseline as a robust efficiency measurement

We find that metrics that only quantify the amount of parallelization within a parallel reasoning trajectory do not accurately reflect true inference-time acceleration obtained when switching from sequential to parallel reasoning. To rigorously assess efficiency gains, we recommend making comparisons directly against *sequential baselines* trained and evaluated under comparable settings.

As summarized in [Table 3](#), we report two quantities: 1) *speedup vs. sequential baseline* ($L_{\text{baseline longest}}/L_{\text{ours longest}}$), and 2) *self-parallelism speedup* ($L_{\text{ours total}}/L_{\text{ours longest}}$). Note that for the sequential baseline, the token latency is the same as the total number of generated tokens.

In three of six benchmarks, the self-parallelism speedup significantly exceeds the speedup measured against sequential baselines. This contrast is informative: AIME 25 shows a clear self-parallelism gain ($1.18\times$) but only a modest $1.03\times$ improvement over the baseline.

Thus, self-parallelism speedup is not a complete efficiency metric. For deployment, the decisive measure is the *critical-path speedup vs. the sequential baseline*. As in other forms of parallelization, APR introduces overhead, including the need to generate outline tokens and additional redundancy because threads operate without visibility into each other. This motivates benchmarking ThreadWeaver against sequential RL on Qwen3-8B in [Table 1](#).

6.3.3 Adding compute makes ThreadWeaver inference faster

Token latency serves as a proxy for critical-path length, but it does not directly measure end-to-end runtime. To verify that reduced critical-path length translates into real wall-clock gains, we run our model with 50 problems from the MATH500 dataset. For each prompt, we run 8 repeated inference passes to reduce variance and report averages across all runs.

When parallelization is disabled, the model executes the entire reasoning trajectory on a single GPU, following standard autoregressive decoding. When parallelization is enabled, the orchestrator distributes parallel threads across four GPUs, allowing concurrent decoding of thread blocks while preserving sequential regions. We set batch size to 1 in both settings. This setting reflects a scenario in which the model is expected to benefit from abundant additional compute when available at inference time.

Table 4 Wall-clock latency comparison between parallel and sequential inference on 50 MATH problems. Latency for each problem is averaged over 8 runs.

Parallelization	Avg Latency (s)
Enabled	142.21
Disabled	162.34
Speedup	1.14×

Table 5 Effect of standard-deviation normalization on AIME 24. Both settings run SFT, self-training, and RL from Qwen3-8B and are evaluated with Avg@32. Length statistics are averaged over all evaluation trajectories.

Setting	Accuracy	Mean Seq. Length	Mean Longest Thread
With Std. Normalization	74.79%	30.1k	18.7k
Mean-Centered Only (Ours)	79.90%	21.1k	16.9k

As shown in [Table 4](#), enabling parallelization reduces wall-clock latency by 1.14×. The observed speedup is smaller than the token-latency speedups reported in earlier sections, which is expected due to overheads in parallelization. Nevertheless, the result confirms that the critical-path reductions learned by the model can lead to end-to-end acceleration once additional compute is provided. This demonstrates that ThreadWeaver can effectively convert additional computational resources into lower inference latency.

6.3.4 Removing standard-deviation normalization in GRPO improves stability

We compare GRPO with and without standard-deviation normalization. Under variance normalization, the scale ρ of the acceleration reward is effectively cancelled whenever all rollouts for a prompt are correct, amplifying the incentive to increase the acceleration ratio $\eta = 1 - \frac{L_{\text{longest}}}{L_{\text{total}}}$ even when this does not reduce the critical path. Empirically, this leads to degraded accuracy and longer trajectories with less meaningful parallelization patterns.

Replacing variance normalization with simple mean-centering preserves the intended relative weight between correctness and acceleration. As shown in [Table 5](#), the mean-centered variant achieves higher accuracy and genuinely shorter critical paths.

6.3.5 High-quality SFT data is critical for model performance

Both our SFT dataset and the Multiverse dataset are constructed from LLM-generated trajectories, but differ in data origin and model alignment. While Multiverse ([Yang et al., 2025b](#)) uses trajectories from s1 ([Muennighoff et al., 2025](#)) generated by DeepSeek-R1 ([DeepSeek-AI, 2025](#)), our dataset is generated directly from Qwen3-8B outputs and then lightly rewritten by GPT-5 ([OpenAI, 2025](#)). To compare these sources fairly, we fine-tune Qwen3-8B on each dataset under identical settings (about 1k samples, 8 epochs). In this ablation, we expand the internal nested parallelization structures in the Multiverse dataset to match the behavior of our model, following how [Yang et al. \(2025b\)](#) curates data for autoregressive baselines. This is not expected to impact accuracy significantly since the accuracy only fluctuates slightly even when all parallelization structures are expanded into sequential structures, as reported in [Yang et al. \(2025b\)](#). To further ablate the impact of data quality, we also fine-tune Qwen3-4B-Base, the backbone used in Parallel-R1 ([Zheng et al., 2025](#)), on our dataset. Parallel-R1 uses its proposed Parallel-GSM8K, curated with DeepSeek-R1-0528-Qwen3-8B ([DeepSeek-AI, 2025](#)) based on GSM8K ([Cobbe et al., 2021a](#)).

As shown in [Table 6](#), training on the Multiverse dataset yields lower performance despite its stronger teacher model. This indicates that dataset-model compatibility is more important than teacher strength in our setting. We hypothesize that DeepSeek-R1 trajectories differ stylistically from Qwen3-8B’s native reasoning patterns, introducing distributional mismatch and limiting generalization. In contrast, our dataset preserves Qwen3-8B’s reasoning semantics while primarily adjusting structural organization to expose parallel reasoning patterns. This allows the model to acquire parallel structure effectively without disrupting its underlying reasoning behavior.

Table 6 Comparison of SFT performance on AIME 24 under different data sources. Our high-quality SFT data yields much stronger performance compared to the datasets curated in Yang et al. (2025b) and Zheng et al. (2025), and RL cannot fully compensate for lower-quality SFT data. * indicate results obtained from Zheng et al. (2025) and are computed with Avg@16. Other results are computed with Avg@32.

Model	AIME24
Qwen3-8B + Multiverse SFT (1000 samples, Yang et al. (2025b))	62.2%
Qwen3-8B + Our first-stage SFT (959 samples)	74.5%
Qwen3-4B-Base + Parallel-GSM8K SFT (7,472 samples, Zheng et al. (2025))	10.6%*
Qwen3-4B-Base + Parallel-GSM8K SFT (7,472 samples, Zheng et al. (2025)) + RL	19.4%*
Qwen3-4B-Base + Our first-stage SFT (959 samples)	32.4%

Table 7 Ablation on the effects of parallel rollout and self-training in RL. Both components contribute to improved reasoning accuracy and reduced token latency. All results in this table are post-RL and are benchmarked on AIME 24.

Parallel Rollout	Self-Training	Accuracy (Avg@32)	Token Latency (\downarrow)
✓	✗	77.9%	17.6k
✗	✓	78.4%	17.3k
✓	✓	79.9%	16.9k

Our SFT model also substantially outperforms Parallel-R1. This is partially due to the fact that Parallel-GSM8k is curated from grade school math dataset GSM8k (Cobbe et al., 2021a), which is likely too simple compared to competition-level math benchmarks. Although Parallel-R1 applies RL, the gains are modest and performance remains much lower. This suggests that *RL cannot fully compensate for lower-quality SFT data*. Both comparisons indicate that high-quality, model-aligned SFT data is a key factor of downstream performance and parallelization ability, whereas additional RL provides only limited gains when the SFT distribution is poorly matched to the target data distribution.

6.3.6 Parallel rollout and self-training improve model performance

We ablate two training components used in our method: *parallel rollout* during RL and *self-training* after the first SFT stage. As described in Table 7, both components contribute to higher accuracy and improved reasoning efficiency.

Without self-training, the model achieves an AIME 24 accuracy of 77.9% and a token latency of 17.6k, which are both sub-par compared to our model. This indicates that training with the 17k high-quality self-training samples does contribute towards improved parallel reasoning capabilities. This is because self-training bridges the gap between samples curated by other LLMs and the reasoning traces that our downstream model, Qwen3-8B, is able to generate, as indicated in Section 4.2.

Parallel rollout during RL allows the policy to directly optimize for parallel execution efficiency while maintaining reasoning correctness, leaving no mismatch between RL rollout and parallel inference during evaluation. Note that self-training can be considered as a form of filtered behavior cloning or STaR (Zelikman et al., 2022b), and since our self-training data is always generated with parallel rollout in the ablations, extensive self-training can be considered as taking part of the role in reinforcement learning. Both techniques are essential for ThreadWeaver to obtain the best performance, with accuracy reaching 79.9% and token latency of 16.9k.

6.4 Discussion: Reward Hacking

A potential concern in RL with parallel rollout is *reward hacking*, where the model artificially inflates the acceleration reward by generating numerous redundant branches. In our experiments, we did not see clear evidence of this behavior. Our model benefits from strong reasoning priors established during supervised fine-tuning on high-quality, parallel-annotated trajectories, which guide it towards meaningful decomposition

rather than excessive branching. Moreover, our RL rollout framework enforces a strict 40k total token limit per trajectory, which also discourages excessive branching (e.g., enumerating thousands of trivial cases). The acceleration ratio reward itself is intentionally conservative, by capping the reward and assigning a small scaling factor to it, so that correctness dominates the overall learning signal.

One solution for this type of reward hacking, which may happen in future search or application, is to apply the length penalty on the token latency. While seemingly more effective than acceleration ratio, it introduces normalization challenges, as the magnitude of the token counts is not naturally comparable to correctness rewards, and poorly normalized penalties can degrade performance by over-penalizing legitimate long reasoning traces. In contrast, the acceleration ratio is normalized relative to each trajectory's structure (i.e., comparing the token latency with total number of tokens) and bounded, maintaining stability and fairness across tasks. For future extensions, more complicated length-based penalties may be applied to the token latency to better account for the potential reward hacking, but in our experiments, the clipped acceleration reward proved sufficient to keep acceleration high without compromising reasoning quality.

6.5 Qualitative Study

To illustrate how ThreadWeaver works in practice, we analyze a trigonometric evaluation example from MATH500 below, in which the model spontaneously forks into multiple reasoning branches to explore distinct solution strategies and joins the reasoning threads to continue the next steps of reasoning. Due to the excessive length of the reasoning trajectories, we use brackets with summarized content in place of bulk reasoning text.

In the first parallelization block, the model explores two distinct approaches to the problem. While the second one reaches a numerical solution, the first one ends up switching to another direction that is more promising. Then in the second parallelization block, the first branch continues to explore the direction that the first branch of the first parallelization block is exploring to get to an exact answer, while the second branch performs self-reflection to double check the numerical answer. These branches parallelize well because they operate on disjoint subgoals that are independent of each other given the shared context. Finally, the main thread integrates the information from all the previous reasoning results and output solutions to the user. This example highlights how ThreadWeaver enables adaptively parallelized exploration and self-reflection within a single inference trajectory. We provide additional error analysis in [Section D.5](#).

ThreadWeaver Example: Trigonometric Expression Evaluation

Prompt:

Evaluate

$$\sin(\arcsin 0.4 + \arcsin 0.5) \cdot \sin(\arcsin 0.5 - \arcsin 0.4).$$

Let's think step by step and output the final answer within `\boxed{}`.

Response:

```
<think>
[Identifies the problem as evaluating sin(arcsin 0.4 + arcsin 0.5) · sin(arcsin 0.5 – arcsin 0.4). Defines A = arcsin(0.4) and B = arcsin(0.5), rewriting the expression as sin(A + B) sin(B – A). Reviews basic sine addition/subtraction formulas and considers expanding directly before looking for a better simplification strategy.]
```

```
<Parallel>
<Goal>
<Outline>1: Use sin(A + B) = sinA cosB + cosA sinB and sin(B – A) = sinB cosA – cosB sinA, then expand the product to get a difference of squares of the form (sinB cosA)^2 – (cosB sinA)^2.</Outline>
<Outline>2: Compute cosA from sinA = 0.4 via cosA = √(1 – sin^2 A) and compute cosB from sinB = 0.5 via cosB = √(1 – sin^2 B), then substitute these values into the simplified-product expression and carry out the arithmetic.</Outline>
</Goal>
<Path>
1: [Attempts expansion, realizes it's cumbersome, and switches to using the product-to-sum identity. Applies sinα sinβ = (cos(α–β) – cos(α+β))/2 with α=A+B and β=A–B, simplifying the product to (cos(2A) – cos(2B)) / 2.]
</Path>
<Path>
2: [Uses numerical approximations for cosA and cosB, then evaluates sin(A+B) and sin(B-A) numerically, confirming the product is about 0.09.]
</Path>
```

```

</Parallel>

<Parallel>
<Goal>
<Outline>1: Use the product-to-sum identity  $\sin(A+B) \cdot \sin(B-A) = [\cos(2B) - \cos(2A)]/2$ , compute  $\cos(2A)$  and  $\cos(2B)$  from  $\sin A$  and  $\sin B$  via  $\cos(2\theta) = 1 - 2 \sin^2 \theta$ , then substitute to get the exact value and check it matches the earlier numerical estimate.</Outline>
<Outline>2: Compute  $A = \arcsin(0.4)$  and  $B = \arcsin(0.5)$  numerically, evaluate  $\sin(A+B)$  and  $\sin(B-A)$  directly, multiply to obtain a numerical approximation, and compare with the symbolic result.</Outline>
</Goal>
<Path>
1: [Computes  $\cos(2A)=0.68$  and  $\cos(2B)=0.5$  using the double-angle identity, plugs into  $(\cos(2A)-\cos(2B))/2$ , and obtains exactly 0.09.]
</Path>
<Path>
2: [Approximates  $A$  and  $B$  numerically, computes  $\sin(A+B)$  and  $\sin(B-A)$ , and confirms the product closely matches 0.09.]
</Path>
</Parallel>

```

[Concludes that both algebraic and numerical methods agree: the value of the expression is 0.09.]

****Final Answer****

\boxed{0.09}

</think>

To evaluate the expression:

$$\sin(\arcsin 0.4 + \arcsin 0.5) \cdot \sin(\arcsin 0.5 - \arcsin 0.4),$$

we begin by introducing substitutions for simplicity:

Let:

- $A = \arcsin(0.4)$, so $\sin A = 0.4$,
- $B = \arcsin(0.5)$, so $\sin B = 0.5$.

Then the expression becomes:

$$\sin(A + B) \cdot \sin(B - A).$$

Step 1: Use Trigonometric Identity

We apply the identity:

$$\sin(A + B) \cdot \sin(B - A) = \frac{\cos(2A) - \cos(2B)}{2}.$$

This identity is derived from the product-to-sum formulas and is particularly useful here since we can compute $\cos(2A)$ and $\cos(2B)$ using the double-angle identity:

$$\cos(2\theta) = 1 - 2 \sin^2(\theta).$$

Step 2: Compute $\cos(2A)$ and $\cos(2B)$

$$\cos(2A) = 1 - 2 \sin^2(A) = 1 - 2(0.4)^2 = 1 - 0.32 = 0.68,$$

$$\cos(2B) = 1 - 2 \sin^2(B) = 1 - 2(0.5)^2 = 1 - 0.5 = 0.5.$$

Step 3: Apply the Identity

$$\sin(A + B) \cdot \sin(B - A) = \frac{\cos(2A) - \cos(2B)}{2} = \frac{0.68 - 0.5}{2} = \frac{0.18}{2} = 0.09.$$

Step 4: Final Answer

Thus, the exact value of the original expression is:

0.09

7 Related work

Test-Time Scaling. Test-time scaling has been shown effective for enhancing LLM performance on a variety of downstream tasks (Wei et al., 2022; Zelikman et al., 2022a; Gandhi et al., 2024; Snell et al., 2025; Team,

2025b; Muennighoff et al., 2025). One significant approach, signified by OpenAI o1 (OpenAI, 2024) and DeepSeek R1 (DeepSeek-AI, 2025), is scaling the length of the trajectories through prolonged chain-of-thought evoked with reinforcement learning. Despite their success, the output sequences generated by these methods are often very long, causing practical challenges. Because the autoregressive decoding employed by most cutting-edge LLMs is sequential, longer outputs cause high latency. As the problems get harder and harder, the time needed to get to an answer increases significantly. For complex problems in the domains of theorem proving and scientific discovery, it is not sustainable to simply increase the latency to achieve improved performance. In contrast to the serialized inference employed in autoregressive LLMs, parallelizing reasoning traces alleviates such issues. Our experiments show that a model trained end-to-end to adaptively allocate computational budgets towards sequential and parallel reasoning can achieve significantly lower latency with the same performance, or improved performance at the same token latency, demonstrating an improved Pareto frontier.

Parallel Reasoning with Self-Consistency. Parallel reasoning through self-consistency has emerged as a widely adopted technique for enhancing the performance of language model inference, providing an orthogonal test-time scaling dimension beyond conventional sequence-length scaling. Instead of following a single reasoning trajectory, multiple independent chains are sampled in parallel and their final answers aggregated to improve robustness (Cobbe et al., 2021b; Wang et al., 2023; Chen et al., 2023). This simple yet effective form of parallelism has been shown to enhance performance across reasoning benchmarks, and is hypothesized to underlie recent large-scale systems such as Gemini DeepThink⁵. More recent works like DeepConf (Fu et al., 2025) further refine this paradigm by introducing confidence-guided aggregation and adaptive consensus mechanisms. However, these methods rely on uncoordinated, independently executed reasoning threads, leading to redundant computation and limited efficiency despite improved answer consistency.

Structured (Agentic) Parallel Reasoning. Another line of work explores structured forms of parallel reasoning through prompting-based agent or tree-of-thought frameworks (Yao et al., 2023; Du et al., 2023; Kim et al., 2024; Grand et al., 2025; Schroeder et al., 2024; Ning et al., 2024; Zhang et al., 2024; Hua et al., 2024; Zhuge et al., 2024; Teng et al., 2025). These approaches coordinate multiple reasoning branches or agents via explicit structures, such as trees, graphs, or modular sub-agents, that decompose complex problems into smaller parallelizable subtasks. While effective in certain domains (e.g., symbolic reasoning, math, or planning), they typically rely on hand-crafted heuristics for task decomposition, control flow, and information aggregation. As a result, they are limited in scalability and generality, since reasoning strategies are externally designed rather than learned. Moreover, most methods operate purely at inference time without model training, leveraging the model’s zero-shot or in-context capabilities rather than learning to reason in parallel end-to-end. Recent agentic formulations such as mixture-of-agents or self-coordinating agent collectives attempt to automate this process, but still depend heavily on prompt-level coordination and lack gradient-based optimization across reasoning branches (Wang et al., 2025; Rodionov et al., 2025; Chen et al., 2025).

Adaptive Parallel Reasoning. Recent efforts have explored training-based methods that enable adaptive parallelization in LLM reasoning. These approaches train models to self-determine when to branch reasoning in parallel and when to continue sequentially, typically controlled by special tokens (e.g., fork-join markers) that mimic multi-threaded execution. Compared to purely sequential reasoning or self-consistency based parallel reasoning, such methods offer the potential to allocate test-time computation more efficiently and adaptively across reasoning paths. PASTA (Jin et al., 2025) applies supervised fine-tuning (SFT) and preference optimization to accelerate instruction-following tasks, while Multiverse (Yang et al., 2025b) fine-tunes on an LLM-rewritten dataset from Muennighoff et al. (2025) to achieve speedups in mathematical reasoning. However, both rely solely on supervised learning and omit reinforcement learning (RL), a key ingredient for test-time scalability. APR (Pan et al., 2025) introduces a complete SFT+RL pipeline, demonstrating that parallel inference can improve both reasoning accuracy and efficiency. However, APR only performed training and evaluation on programmatically synthesized datasets. Concurrent to our work, Parallel-R1 (Zheng et al., 2025) also incorporates RL-based optimization, but its SFT data covers only low-complexity problems, and its reasoning traces remain substantially shorter and less capable than those of standard sequential reasoning baselines (Yang et al., 2025a). In comparison, our approach adapts existing reasoning models rather than

⁵<https://blog.google/products/gemini/gemini-2-5-deep-think/>

training from scratch, maintaining their performance while achieving significantly lower latency at equivalent accuracy or improved accuracy under the same token budget. Furthermore, unlike PASTA, Multiverse, and Parallel-R1, our approach requires no modification to the inference engine, only a lightweight client-side wrapper that issues parallel requests when instructed by the model.

8 Conclusion

We introduce ThreadWeaver, an adaptive parallel reasoning framework that addresses three obstacles that have limited prior adaptive parallel reasoning in practice: 1) a two-stage parallel trajectory generator that converts long chain-of-thought traces into high-quality trajectories with parallel annotations at scale, 2) a trie-based training inference co-design that aligns supervision with the deployment-time state machine and yields a single hybrid model that can run in either purely sequential or parallel mode without changing serving infrastructure, and 3) a parallel-aware RL framework, P-GRPO, that trains our model for improved answer accuracy and acceleration. Empirically, ThreadWeaver achieves up to $1.53\times$ speedup in terms of token latency on math problems compared to sequential baselines, achieving a new Pareto frontier and laying the groundwork for efficient parallelized solutions to complex real-world problems.

A future direction is enabling the model to reason about the available resources on which it is deployed, such as the number of available GPUs and network topology, so it can adaptively decide how many threads to spawn and how to distribute work. Training could incorporate diverse hardware profiles, with feedback grounded in real rollout measurements. Another promising direction is to extend parallelization beyond reasoning to interactions with environments, enabling multiple agents to work in parallel in domains such as software engineering and scientific research. Achieving this will require new coordination mechanisms that allow agents to share information, avoid conflicts, and remain aligned toward a common goal.

Acknowledgements

We thank Jiayi Pan and Abhimanyu Dubey for helpful early discussions. We thank Xinyu Yang and Beidi Chen for answering our questions about the Multiverse paper and codebase. We thank Robert Nishihara, Richard Liaw, Jiajun Yao, and the Anyscale team for providing us with an early implementation of Ray with IPv6 support, even though we ultimately adopted a different approach due to timing constraints. We thank Vignesh Ramanathan and Natalia Neverova for their support of this project.

References

- Chenxin An, Zhihui Xie, Xiaonan Li, Lei Li, Jun Zhang, Shansan Gong, Ming Zhong, Jingjing Xu, Xipeng Qiu, Mingxuan Wang, and Lingpeng Kong. Polaris: A post-training recipe for scaling reinforcement learning on advanced reasoning models, 2025. <https://hkunlp.github.io/blog/2025/Polaris>.
- Anthropic. Introducing claude sonnet 4.5. <https://www.anthropic.com/news/clause-sonnet-4-5>, September 2025.
- Xinyun Chen, Renat Aksitov, Uri Alon, Jie Ren, Kefan Xiao, Pengcheng Yin, Sushant Prakash, Charles Sutton, Xuezhi Wang, and Denny Zhou. Universal self-consistency for large language model generation, 2023. <https://arxiv.org/abs/2311.17311>.
- Yongchao Chen, Jiefeng Chen, Rui Meng, Ji Yin, Na Li, Chuchu Fan, Chi Wang, Tomas Pfister, and Jinsung Yoon. Tumix: Multi-agent test-time scaling with tool-use mixture, 2025. <https://arxiv.org/abs/2510.01279>.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021a.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *ArXiv*, 2110.14168, 2021b.
- DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in llms via reinforcement learning. *ArXiv*, abs/2501.12948, 2025.
- Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.
- Yilun Du, Shuang Li, A. Torralba, J. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *ICML*, 2023.
- Yichao Fu, Xuewei Wang, Yuandong Tian, and Jiawei Zhao. Deep think with confidence, 2025. <https://arxiv.org/abs/2508.15260>.
- Kanishk Gandhi, Denise HJ Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah Goodman. Stream of Search (SoS): Learning to search in language. In *COLM*, 2024.
- Gabriel Grand, Joshua B Tenenbaum, Vikash K Mansinghka, Alexander K Lew, and Jacob Andreas. Self-steering language models. *arXiv preprint arXiv:2504.07081*, 2025.
- Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*, 2024.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.
- Wenyue Hua, Mengting Wan, Shashank Vadrevu, Ryan Nadel, Yongfeng Zhang, and Chi Wang. Interactive speculative planning: Enhance agent efficiency through co-design of system and user interface. *arXiv preprint arXiv:2410.00079*, 2024.
- Tian Jin, Ellie Y Cheng, Zack Ankner, Nikunj Saunshi, Blake M Elias, Amir Yazdanbakhsh, Jonathan Ragan-Kelley, Suvinay Subramanian, and Michael Carbin. Learning to keep a promise: Scaling language model decoding parallelism with learned asynchronous decoding. *arXiv preprint arXiv:2502.11517*, 2025.
- Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *ICML*, 2024.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in neural information processing systems*, 35:3843–3857, 2022.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepScaler: Surpassing O1-preview with a 1.5b model by scaling rl. <https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2>, 2025. Notion Blog.

Mathematical Association of America. American mathematics competitions 2023. <https://huggingface.co/datasets/AI-MO/aimo-validation-amc>, 2023.

Mathematical Association of America. American invitational mathematics examination 2024. https://artofproblemsolving.com/wiki/index.php/American_Invitational_Mathematics_Examination, 2024.

Mathematical Association of America. American invitational mathematics examination 2025. https://artofproblemsolving.com/wiki/index.php/American_Invitational_Mathematics_Examination, 2025.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting LLMs for efficient parallel generation. In *ICLR*, 2024.

OpenAI. Learning to reason with LLMs, 2024. <https://openai.com/index/learning-to-reason-with-lmss/>.

OpenAI. Introducing gpt-5. <https://openai.com/index/introducing-gpt-5/>, August 2025.

Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning adaptive parallel reasoning with language models. *arXiv preprint arXiv:2504.15466*, 2025.

Qwen, ;, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. <https://arxiv.org/abs/2412.15115>.

Gleb Rodionov, Roman Garipov, Alina Shutova, George Yakushev, Vage Egiazarian, Anton Sinitzin, Denis Kuznedelev, and Dan Alistarh. Hogwild! Inference: Parallel LLM generation via concurrent attention. *arXiv*, 2504.06261, 2025.

Philip Schroeder, Nathaniel Morgan, Hongyin Luo, and James Glass. Thread: Thinking deeper with recursive spawning. *arXiv preprint arXiv:2405.17402*, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. <https://arxiv.org/abs/1707.06347>.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, 2402.03300, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *ICLR*, 2025.

Gemini 2.5 Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025a. <https://arxiv.org/abs/2507.06261>.

Kimi Team. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv*, 2501.12599, 2025b.

Fengwei Teng, Zhaoyang Yu, Quan Shi, Jiayi Zhang, Chenglin Wu, and Yuyu Luo. Atom of thoughts for Markov LLM test-time scaling. *arXiv*, 2502.12018, 2025.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities. In *ICLR*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *ICLR*, 2023.

- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengan Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.
- Xinyu Yang, Yuwei An, Hongyi Liu, Tianqi Chen, and Beidi Chen. Multiverse: Your language models secretly decide how to parallelize and merge generation. *arXiv preprint arXiv:2506.09991*, 2025b.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS*, 2023.
- Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.
- Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STar: Bootstrapping reasoning with reasoning. In *NeurIPS*, 2022a.
- Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022b.
- Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Arik. Chain of agents: Large language models collaborating on long-context tasks. *NeurIPS*, 2024.
- Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.
- Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sqlang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.
- Tong Zheng, Hongming Zhang, Wenhao Yu, Xiaoyang Wang, Xinyu Yang, Runpeng Dai, Rui Liu, Huiwen Bao, Chengsong Huang, Heng Huang, et al. Parallel-r1: Towards parallel thinking via reinforcement learning. *arXiv preprint arXiv:2509.07980*, 2025.
- Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbulin, and Jürgen Schmidhuber. Language agents as optimizable graphs. *ICML*, 2024.

Appendix

A Derivations for P-GRPO

A.1 From clipped GRPO to the score-function objective

For completeness, we first establish that P-GRPO with strictly on-policy sampling reduces to a score-function (REINFORCE) estimator with a baseline. Define the PPO-style clipped GRPO surrogate over a batch \mathcal{B} with group size k :

$$\mathcal{J}_{\text{clip}}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{p \in \mathcal{B}} \frac{1}{k} \sum_{i=1}^k \mathbb{E}_{\tau_p^{(i)} \sim \pi_{\theta_{\text{old}}}} \left[\min(r_{\theta}(\tau_p^{(i)}) A_{p,i}, \text{clip}(r_{\theta}(\tau_p^{(i)}), 1 - \epsilon, 1 + \epsilon) A_{p,i}) \right], \quad (10)$$

where $A_{p,i}$ is the advantage computed via GRPO and the importance ratio is:

$$r_{\theta}(\tau) = \frac{\pi_{\theta}(\tau)}{\pi_{\theta_{\text{old}}}(\tau)}. \quad (11)$$

The gradient of the unclipped objective with respect to θ is:

$$\nabla_{\theta} \mathbb{E}[r_{\theta}(\tau) A] = \mathbb{E}[A \nabla_{\theta} r_{\theta}(\tau)] = \mathbb{E}[A r_{\theta}(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau)]. \quad (12)$$

In our setting, we perform a single gradient update immediately after data collection (fully on-policy), implying $\theta_{\text{old}} = \theta$. Consequently, $r_{\theta}(\tau) = 1$ identically, rendering the clipping operation inactive. Substituting $r_{\theta}(\tau) = 1$ into Equation (12) yields the standard score-function gradient:

$$\nabla_{\theta} \mathcal{J}_{\text{P-GRPO}}(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} [A(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau)]. \quad (13)$$

A.2 Factorization of Parallel Reasoning Trajectories

To implement Equation (13) efficiently for tree-structured parallel reasoning, we factorize the probability of the trajectory. Let a parallel trajectory $\tau_p^{(i)}$ be represented as a sequence of M_i disjoint generation segments (or units) produced by the state machine:

$$\tau_p^{(i)} = ((\text{cont}^{(i,1)}, \text{comp}^{(i,1)}), \dots, (\text{cont}^{(i,M_i)}, \text{comp}^{(i,M_i)}))$$

Here, each unit m consists of a context $\text{cont}^{(i,m)}$ (which may include the prompt and previously generated segments) and a newly generated completion $\text{comp}^{(i,m)}$. Without loss of generality, in this derivation, we ignore tokens that we do not apply loss on (e.g., the user prompt and the tokens inserted by the state machine orchestrator). Under this assumption, the set of all completion segments $\{\text{comp}^{(i,m)}\}_{m=1}^{M_i}$ forms a partition of all tokens generated by the model in the trajectory $\tau_p^{(i)}$.

By the chain rule of probability, the log-probability of the entire trajectory is the sum of the log-probabilities of its generated tokens. Since each completion segment $\text{comp}^{(i,m)}$ is generated conditioned on its specific history $\text{cont}^{(i,m)}$, we have:

$$\log \pi_{\theta}(\tau_p^{(i)}) = \sum_{m=1}^{M_i} \log \pi_{\theta}(\text{comp}^{(i,m)} | \text{cont}^{(i,m)}). \quad (14)$$

This factorization holds regardless of the parallel structure. For example, a shared `<Outline>` segment is simply the completion of the first unit (conditioned on the prompt), while subsequent `<Thread>` segments are completions of later units (conditioned on the prompt and the Outline). The independence of parallel threads is naturally captured by the fact that their contexts branch from the same parent node in the trie.

A.3 Broadcasted Credit Assignment

We now show that broadcasting the trajectory-level advantage to all tokens in all threads is mathematically justified by relating it to the gradient of the total trajectory loss. Substituting the factorization from [Equation \(14\)](#) into the gradient objective in [Equation \(13\)](#):

$$\begin{aligned} \nabla_{\theta} \mathcal{J}_{\text{P-GRPO}}(\theta) &= \mathbb{E}_{\tau_p^{(i)} \sim \pi_{\theta}} \left[A_{p,i}^{\text{P-GRPO}} \nabla_{\theta} \log \pi_{\theta}(\tau_p^{(i)}) \right] \\ &= \mathbb{E}_{\tau_p^{(i)} \sim \pi_{\theta}} \left[A_{p,i}^{\text{P-GRPO}} \nabla_{\theta} \left(\sum_{m=1}^{M_i} \log \pi_{\theta}(\text{comp}^{(i,m)} \mid \text{cont}^{(i,m)}) \right) \right] \\ &= \mathbb{E}_{\tau_p^{(i)} \sim \pi_{\theta}} \left[\sum_{m=1}^{M_i} \left(A_{p,i}^{\text{P-GRPO}} \nabla_{\theta} \log \pi_{\theta}(\text{comp}^{(i,m)} \mid \text{cont}^{(i,m)}) \right) \right]. \end{aligned} \quad (15)$$

[Equation \(15\)](#) demonstrates that the gradient of the full trajectory objective is simply the sum of the gradients for each completion segment, weighted by the *same* scalar trajectory advantage $A_{p,i}^{\text{P-GRPO}}$.

In our implementation, we approximate the expectation using a Monte Carlo estimate over the batch \mathcal{B} . The broadcasting operation described in [Section 5.2](#) assigns the scalar $A_{p,i}^{\text{P-GRPO}}$ to every token in every completion $\text{comp}^{(i,m)}$. When we compute the loss on these tokens weighted by $A_{p,i}^{\text{P-GRPO}}$, we are computing exactly the inner term of the summation in [Equation \(15\)](#). Thus, P-GRPO’s advantage broadcasting is not a heuristic approximation. It is the policy gradient estimator for the parallel trajectory reward structure.

B Implementation Details

We provide implementation details for ThreadWeaver training pipeline.

Data Preparation. Training prompts are sourced from the Polaris 53k dataset ([An et al., 2025](#)), which contains 53,291 high-quality math reasoning problems. Each prompt contains a question and a reference answer. We perform initial inference with Qwen3-8B to obtain a single reasoning trajectory per prompt. In all generated trajectories, 36,296 (68.1%) contain both a correct final answer (validated by symbolic evaluation) and a valid reasoning structure. These form the base candidate trajectory pool for SFT.

Trie-based Sequence Merging In addition to the single branching case in the example provided by [Figure 4](#), our trie-based sequence merging also works on trajectories that contain multiple `<Parallel>` blocks interleaved with sequential reasoning segments. We explain how trie-based sequence merging works with multiple `<Parallel>` blocks in detail.

As illustrated in [Figure 6](#), we follow the same three steps as in the single-block case: extract all `<context, completion>` units that the inference state machine would issue, insert their token sequences into a shared prefix tree, and traverse the trie to obtain a single packed sequence with an ancestor-only attention mask. Loss is still applied only to completion tokens, so the total number of supervised tokens matches training on each unit independently. Because the model also supports pure autoregressive decoding, we additionally treat the full trajectory as a single `<context, completion>` unit, with the user prompt as the context and the full model completion as the completion. For simplicity, we omit this full trajectory unit in [Figure 4](#) and [Figure 6](#), as including the unit is only to allow the model to perform sequential reasoning with standard autoregressive decoding.

Supervised Fine-tuning (SFT). From the 36,296 valid trajectories, we sample 1,000 for rewriting using GPT-5 ([OpenAI, 2025](#)). Each sample is rewritten with a structured instruction prompt to improve readability and ensure canonical parallelization, while preserving the exact step order and final boxed answer. After automatic filtering for format and semantic alignment, we use the remaining 959 high-quality rewritten trajectories as the training data for SFT.

SFT is performed for 8 epochs with the following hyperparameters:

Step 1: Extracting all <context, completion> pairs from a training example

What is the sum of the roots of quadratic equation $x^2 - 5x + 6 = 0$? Think step by step and put your answer in `boxed()`.

<think>

We want to solve $x^2 - 5x + 6 = 0$. We can either factor or use the quadratic formula. First, we will use the quadratic formula to get both solutions.

<Parallel>

<Outlines>

<Outline>1: Compute the discriminant $\Delta = b^2 - 4ac$.

<Outline>2: Compute the denominator $2a$ and set up the formula.

</Outlines>

<Thread>1: Here $a = 1, b = -5, c = 6$.

$\text{Compute } \Delta = (-5)^2 - 4 \cdot 1 \cdot 6 = 25 - 24 = 1.$

<Thread>2: We have $2a = 2 = 1$.

So the quadratic formula is

$$x = \frac{-b \pm \sqrt{\Delta}}{2a} = \frac{5 \pm \sqrt{1}}{2}.$$

</Thread>

</Parallel>

Now substitute $\Delta = 1$ into the formula and simplify the two roots.

<Parallel>

<Outlines>

<Outline>1: Compute the plus root x_1 .

<Outline>2: Compute the minus root x_2 .

</Outlines>

<Thread>1: $x_1 = (5 + \sqrt{1}) / 2 = (5 + 1)/2 = 6/2 = 3$.

So one solution is 3 .

<Thread>2: $x_2 = (5 - \sqrt{1}) / 2 = (5 - 1)/2 = 4/2 = 2$.

So the other solution is 2 .

</Thread>

</Parallel>

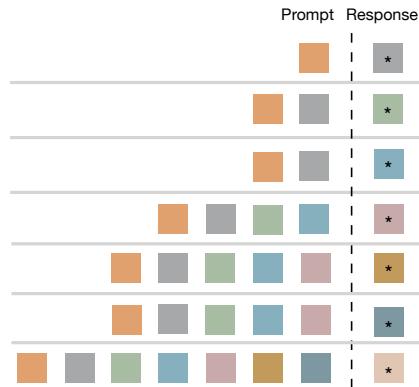
We have found both roots $x = 2$ and $x = 3$.

The sum of the two roots is $2 + 3 = 5$.

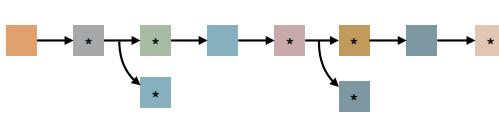
Therefore, the solution is `boxed(5)`.

</think>

The sum of the roots for quadratic $x^2 - 5x + 6 = 0$ is `$boxed(5)$`.



Step 2: Prefix-Tree Building



Step 3: Tree Flattening to Single Sequence

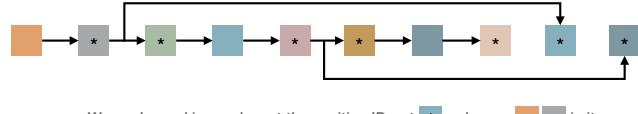


Figure 6 Trie-based sequence merging with more than one parallel block. We still build a prefix-tree (trie) based on the \langle context, completion \rangle pairs from the training example and then flatten the tree into one packed training sequence with an ancestor-only attention mask.

- **Optimizer:** AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and weight decay 0.01.
 - **Learning rate:** initial 1×10^{-5} with 5% warmup and cosine decay.
 - **Batch size:** 16.
 - **Sequence length:** maximum 40,960 tokens.
 - **Loss:** standard cross-entropy on the tokens with loss mask specified in Section 3.

Self-training. We apply the SFT-trained model to the full 53k Polaris prompt set to generate new reasoning trajectories. Each trajectory is automatically scored by a dual-criteria filter:

1. **Answer correctness:** whether the final boxed expression matches the ground truth, using symbolic equivalence checks implemented via SymPy.
 2. **Structural validity:** whether all <Parallel> and </Parallel> markers form a valid structure, ensuring no partial threads or misuse of special tokens.

The trajectories generated with this process are used for the second-stage fine-tuning.

The self-training SFT stage uses identical hyperparameters to the initial SFT, except we only run for 1 epoch.

Reinforcement Learning. The final stage employs the proposed **Parallel-GRPO (P-GRPO)** algorithm. Parallel rollouts are performed so that each of the prompts is sampled 8 times in each batch, and sequence-level advantages are normalized by group mean within each prompt. Additional hyperparameters are listed as follows:

- **Optimizer:** Adam with learning rate 5×10^{-6} , no weight decay.

Table 8 Statistics for the three training trajectory datasets. “Sequential” are raw Qwen3–8B trajectories subsampled from the 36k trajectories with the correct answers; “First-stage” are GPT–5–rewritten parallel traces; “Self-train” are trajectories generated by the model trained with data in the first stage and filtered as in section 4.2. Parallel and acceleration ratios are computed per trajectory then averaged.

	Sequential Trajectories (sampled)	First-stage	Self-train
Size (format-correct)	1000	959	17,491
Avg. parallel ratio	0.00	0.77	0.68
Avg. longest seq. length	10,377	8,278	9,153
Avg. total tokens	10,377	11,143	11,176
Self-parallelism speedup	1.00	1.35	1.22
Avg. # parallel blocks	0.0	6.26	4.00
Avg. parallel path length	—	541	969
Avg. tokens per parallel block	—	1,542	2,458
Avg. threads per parallel block	—	2.58	2.46

- **Batch size:** 128 (8 rollouts \times 16 prompts).
- **KL penalty:** No KL penalty is applied.
- **Temperature:** 1.0 during rollout and at evaluation.
- **Precision:** bf16.

We run the training for 350 optimization steps. We use vLLM (Kwon et al., 2023) as our parallel inference backend. We leverage fully-sharded data parallel (FSDP, Zhao et al. (2023)) in distributed training.

Evaluation Setup. Evaluation is conducted across six datasets: AIME24 (Mathematical Association of America, 2024), AIME25 (Mathematical Association of America, 2025), AMC23 (Mathematical Association of America, 2023), MATH500 (Hendrycks et al., 2021), Minerva Math (Lewkowycz et al., 2022), and OlympiadBench (He et al., 2024). Inference is conducted with total number of tokens set to 40k across the parallel branches.

C Dataset Statistics

We report core statistics of the three trajectory datasets used in ThreadWeaver training: **1**) the original sequential Qwen3–8B trajectories on Polaris–53K, **2**) the dataset with parallel annotations processed by GPT-5 (OpenAI, 2025) and used in the first stage of training (Section 4.1), and **3**) the larger self-training dataset generated by the model itself (Section 4.2).

The original trajectories are long and purely sequential, with no acceleration from parallelization. Rewriting with GPT-5 introduces substantial parallel structure: about 77% of tokens lie in parallel blocks, and the critical path is on average about 26% shorter than the total token count, despite a slight increase in overall length. This corresponds to an idealized self-parallelism speedup of roughly 1.35 \times (computed by dividing the token latency with total number of tokens averaged across the dataset) and provides strong supervision for the fork-join format. We also observe rich structural annotations, with about 2.6 threads per parallel block on average

As discussed in Section 4, however, the rewritten traces are produced post hoc from complete solutions and do not always match how the model naturally decomposes problems at inference time: the strong teacher can rely on global hindsight and sometimes parallelizes by skipping or compressing intermediate steps that the student needs. To close this distribution gap, we perform self-training with the trajectories extracted by the model obtained from the first stage under our proposed parallel inference state machine, then filter trajectories to keep only structurally valid and answer-correct samples, with length and acceleration constraints. The resulting self-training corpus is much larger (17,491 trajectories). Since self-training only takes parallel trajectories that the smaller student model is able to generate ad hoc during reasoning, where it does not have full view of the

Table 9 Stagewise performance on AIME 24. *Format correctness* checks proper closure of all opened `Parallel` blocks and well-formed thread markers (a proxy for structural stability). Common reasons for format error are truncation under token budgets and early stopping without closing the parallel block. Correctness of the final answer depends only on the last `\boxed{}`.

Model	Format Correctness	Accuracy (Avg@32)
Qwen3-8B (Baseline without SFT)	–	76.0%
Qwen3-8B + 1st SFT (959 SFT examples)	56.4%	74.5%
Qwen3-8B + 1st SFT + RL	41.9%	77.9%
Qwen3-8B + self-training (17k SFT examples)	77.0%	74.0%
Qwen3-8B + self-training + RL	72.4%	79.9%

Table 10 Ablation of sequential–parallel cold-start mixing strategies during SFT on AIME 24. “No sequential mixture cold start” uses only parallel data in SFT. “Multiverse-style mixing recipe” follows a non-gradual schedule that switches between mostly sequential, mixed, and mostly parallel phases. “Linear mixing recipe” uses an eight-stage curriculum that linearly increases the proportion of parallel data across epochs. Neither strategy improves over the parallel-only SFT setting, suggesting that additional sequential data does not help Qwen3-8B transition to parallel reasoning once it already has strong sequential capabilities.

Model	Accuracy (Avg@32)
Qwen3-8B (No sequential mixture cold start)	74.5%
Qwen3-8B (Cold start with Multiverse-style mixing recipe)	73.4%
Qwen3-8B (Cold start with linear mixing recipe)	73.1%

overall trajectories except for the tokens it has generated, self-training reduces the self-parallelism speedup to $1.22\times$ while remaining strongly parallel, and better reflects what the model itself can reliably parallelize.

Together, our dataset from the two-stage data pipeline provides robust training signal for both supervised fine-tuning and prepares the model for the subsequent reinforcement learning.

D Additional Results

D.1 Performance Across Training Stages

We evaluate performance on AIME 24 across successive training stages, measuring both *accuracy* (Avg@32) and *format correctness*, which reflects whether all opened `Parallel` blocks are properly closed. Format correctness serves as a proxy for structural stability, though it does not directly affect answer accuracy since the final result depends only on the last `\boxed{}` expression.

As shown in [Table 9](#), accuracy drops after the first SFT on 959 examples (74.5%), with limited format correctness (56.4%). This indicates that small-scale supervised data is insufficient for mastering the parallel reasoning. Expanding to 17k examples via self-training substantially improves format correctness (77.0%) while maintaining similar accuracy, showing that model-generated data helps the model internalize structural consistency. Reinforcement learning (RL) further improves accuracy in both cases (77.9% and 79.9%), but format correctness decreases slightly when initialized from the smaller dataset. This is expected since RL directly optimizes outcome and efficiency rather than explicit structural closure. Overall, self-training yields a more stable structural foundation, enabling RL with parallel rollout to achieve higher accuracy and lower token latency.

D.2 Cold-start with Sequential Data Mixture

Multiverse ([Yang et al., 2025b](#)) proposed to cold-start with a mixture of sequential and parallel data in order to reduce the degradation that can occur when directly fine-tuning on parallel reasoning trajectories. We also

ablated such cold-start strategies in our SFT stage, using a dynamic mixture between two datasets: a purely sequential dataset \mathcal{D}_{seq} built from autoregressive Qwen3-8B trajectories, and a parallel dataset \mathcal{D}_{par} built from our parallelized trajectories rewritten from \mathcal{D}_{seq} . We then constructed curriculum schedules over SFT epochs that control how often each dataset is sampled. Concretely, we considered two families of curricula: a *staged* schedule that switches in a few coarse phases, following [Yang et al. \(2025b\)](#), and a gradual schedule that increases the share of parallel data linearly over eight epochs. Both variants are trained for 8 epochs, following the training schedule of ThreadWeaver, for fair comparison. Only SFT is conducted for this comparison to isolate the effect of each cold-start strategy.

Staged mixture (Multiverse-style mixing recipe). For the staged schedule, we follow [Yang et al. \(2025b\)](#) and organize SFT into three phases:

- **Phase 1: fully sequential phase.** The model is trained for 2 epochs using only \mathcal{D}_{seq} .
- **Phase 2: mixed phase.** The model is trained for 4 epochs on mixtures of sequential and parallel data, implemented as four one-epoch subphases with different fixed mixing ratios.
- **Phase 3: fully parallel phase.** The model is trained for 2 epochs using only \mathcal{D}_{par} .

Conceptually, the curriculum can be summarized as

$$\underbrace{\mathcal{D}_{\text{seq}}}_{\text{fully sequential phase}} \rightarrow \underbrace{\alpha \mathcal{D}_{\text{seq}} + (1 - \alpha) \mathcal{D}_{\text{par}}}_{\text{mixed phase}} \rightarrow \underbrace{\mathcal{D}_{\text{par}}}_{\text{fully parallel phase}},$$

where $\alpha \in (0, 1)$ controls the fraction of sequential data. In Multiverse’s implementation, α is not a single constant but is implicitly determined by how the two datasets are sliced and concatenated. The four mixed subphases correspond to:

- A first mixed epoch that uses 90% \mathcal{D}_{seq} and 10% \mathcal{D}_{par} (effectively $\alpha = 0.9$)
- Two middle mixed epochs that each combine 50% \mathcal{D}_{seq} and 50% \mathcal{D}_{par} (effectively $\alpha = 0.5$)
- A final mixed epoch that uses 10% \mathcal{D}_{seq} and 90% \mathcal{D}_{par} (effectively $\alpha = 0.1$)

Intuitively, Phase 2 therefore sweeps from mostly sequential, through approximately balanced mixtures, to mostly parallel before transitioning to the fully parallel Phase 3.

Gradual mixture (linear mixing recipe). In the second variant, we construct a smooth curriculum over eight stages, each corresponding to roughly one epoch. We again treat \mathcal{D}_{seq} as the compatible supervised signal and \mathcal{D}_{par} as the target format. At epoch $k \in \{1, \dots, 8\}$ we set a target fraction for parallel data

$$\lambda_k = 0.125 \times k,$$

and sample a batch that contains a fraction λ_k from \mathcal{D}_{par} and $(1 - \lambda_k)$ from \mathcal{D}_{seq} . This produces the progression

$$12.5\% \text{ parallel} \rightarrow 25\% \rightarrow 37.5\% \rightarrow 50\% \rightarrow 62.5\% \rightarrow 75\% \rightarrow 87.5\% \rightarrow 100\% \text{ parallel},$$

so the model starts in a regime that is almost fully sequential and ends in a purely parallel regime, with the proportion of parallel examples changing smoothly at each stage. Within each stage we shuffle both datasets and sample without additional bias, so the only variable that changes across epochs is the global ratio between sequential and parallel trajectories.

In both settings, the goal is to cushion the transition from sequential to parallel reasoning, either by a small number of coarse phases (staged mixture) or by a fine-grained eight-epoch ramp (gradual mixture). However, as shown in [Table 10](#), neither curriculum yielded clear benefits over directly training on parallel data without any sequential mixture. We hypothesize that this is because Qwen3-8B is already a strong reasoning model with robust sequential behavior. When sequential data is continually mixed in, the model receives a strong training signal that reinforces its existing autoregressive reasoning style, which slows down the adaptation to the new parallel format. In other words, the mixed curricula make it harder for the model to fully commit to parallel structure, so the additional sequential examples do not translate into better performance in our setting, despite their success as a cold-start recipe in Multiverse.

Table 11 Performance of base models on AIME 24 after supervised fine-tuning with two data sources. “AR SFT” denotes fine-tuning on autoregressive Qwen3-8B trajectories (1k samples); “Our first-stage SFT” denotes GPT-5-processed parallel trajectories (959 samples). Without prior instruction-tuning, parallel SFT on base models lags behind AR SFT, indicating that existing reasoning and instruction-following capabilities are important for effective parallelization.

Model	Accuracy (Avg@32)
Qwen3-4B-Base + AR SFT (1k SFT examples)	37.4%
Qwen3-4B-Base + Our first-stage SFT (959 SFT examples)	32.4%
Qwen3-8B-Base + AR SFT (1k SFT examples)	55.1%
Qwen3-8B-Base + Our first-stage SFT (959 SFT examples)	45.5%

D.3 ThreadWeaver Reuses Existing Reasoning Capabilities for Parallel Reasoning

We start from a reasoning model, Qwen3-8B, for our main experiments. In contrast, we would also like to explore if parallel reasoning works if we start from a base model. We experiment with Qwen3-4B-Base and Qwen3-8B-Base. AR SFT is performing SFT with the Qwen3-8B trajectories. Our first-stage SFT indicates performing SFT with the Qwen3-8B trajectories, processed by our pipeline through GPT-5. Note that our pipeline discards some trajectories, resulting in 959 samples.

As shown in [Table 11](#), in contrast to similar performance obtained by Qwen3-8B vs Qwen3-8B after SFT, the performance of models fine-tuned with parallel SFT drops significantly, both on Qwen3-4B-Base and Qwen3-8B-Base. Since the base models have not been post-trained, the models do not possess strong instruction following capabilities after a small amount of parallel SFT, causing them to not be able to generate high-quality outlines and follow these outlines to solve the problem in several threads. The limited instruction following capabilities cause the degraded performance when parallel reasoning is instilled into the model. Existing reasoning capabilities and instruction following capabilities are needed so that a small amount of parallel SFT could align the model to perform parallel reasoning well. Since the difference between AR SFT and our parallel SFT is too large, we do not proceed to subsequent stages of training on base models.

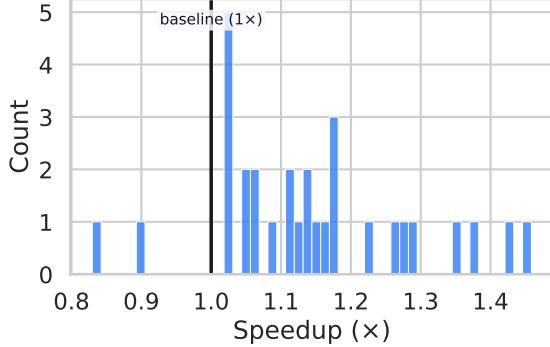
Note that Parallel-R1 ([Zheng et al., 2025](#)), a concurrent work to ours, also uses Qwen3-4B-Base as their base model and demonstrates parallel reasoning by running SFT and RL. However, their reported performance (10.6% with SFT, 19.4% with SFT + RL, as reported in [Zheng et al. \(2025\)](#)) is much lower than the performance of our SFT result with parallel SFT (32.4%), despite the fact that our parallel SFT lags behind AR SFT on base models. The comparison shows that our datasets surpass Parallel-R1 in terms of performance after parallel SFT.

D.4 Per-problem Speedup Without Filtering

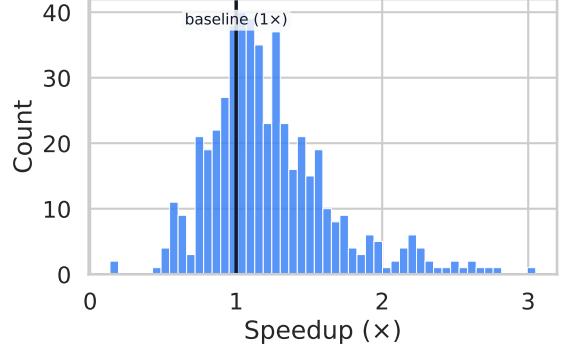
We also study per-problem speedup without filtering for correctness, as shown in [Figure 7](#). In this setting every rollout is included, so the distributions reflect the speedup that a user would observe when running each model end to end, irrespective of whether the final answer is correct. While the maximum values in the upper tail are harder to interpret, since they could capture a potential case that our model terminates earlier incorrectly while the autoregressive baseline continues reasoning and succeeds, the overall distributions remain very similar to those in [Figure 5](#). In particular, the bulk of the mass still lies to the right of the $1.0 \times$ reference line and the shape of the histograms changes only slightly. Empirically, this indicates that the measured acceleration is not primarily driven by sacrifices in accuracy, but instead reflects real reductions in token latency relative to the sequential baseline.

D.5 Error Analysis: Repetition in Branches

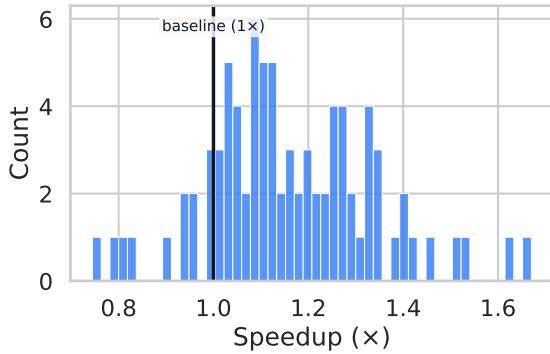
We show a failure case below. Although the answer is correct, this is a failure case because the purportedly parallel threads do not provide genuinely complementary reasoning and instead largely duplicate the same computation: both compute the exponent of 5 in $42!$ via the standard floor-sum formula and both verify the method on smaller factorials such as $10!$ and $25!$, so the parallel structure adds little informational value.



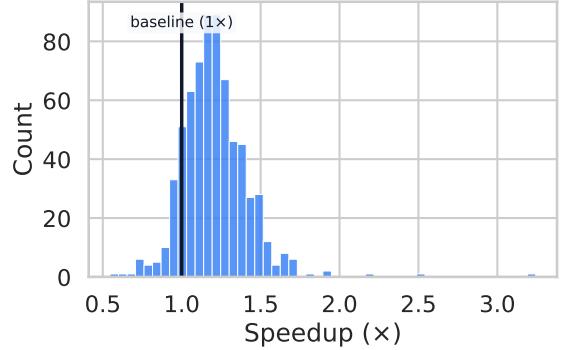
(a) AIME24 (Mean speedup: 1.14 \times)



(b) MATH500 (Mean speedup: 1.23 \times)



(c) AMC23 (Mean speedup: 1.17 \times)



(d) OlympiadBench (Mean speedup: 1.20 \times)

Figure 7 Per-problem speedup distributions on math benchmarks without filtering for correctness. The speedup is the ratio between the token latency of the autoregressive baseline and the token latency of ThreadWeaver. A vertical reference line at 1.0 \times marks parity with the sequential baseline. In contrast to figure 5, these histograms include all rollouts rather than restricting to samples with correct final answers. However, taking both sets of histograms into account, we show that ThreadWeaver achieves substantial speedup on most problems no matter whether correctness filtering is performed.

Moreover, the alignment between outlines and threads is not respected: rather than having one path focus on the main calculation for 42! and the other confined to checking the method on smaller examples, each path blends these tasks and performs both the primary computation and the auxiliary validations. This redundancy and mixing of roles undermines the intended division of labor across paths and illustrates why the example does not realize the benefits of structured parallel reasoning, indicating the room for improvements in the instruction following capabilities.

ThreadWeaver Example (Failure Case): Number of trailing zeros in 42!

Prompt:

How many zeroes are at the end of 42! (42 factorial)? (Reminder: The number $n!$ is the product of the integers from 1 to n . For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.) Let's think step by step and output the final answer within `\boxed{}`.

Response:

```
<think>
[Recognizes that trailing zeros come from factors of 10 = 2·5 in the prime factorization, and that in factorials there are
many more 2s than 5s, so the number of trailing zeros is the exponent of 5 in 42!. Plans to use the standard floor-sum
formula.]
```

```
<Parallel>
<Outlines>
<Outline>1: Use the formula for the exponent of 5 in 42!: sum floor(42/5^k) over k = 1, 2, ... until the terms are
zero.</Outline>
```

```

<Outline>2: Validate the method on smaller factorials such as 10! and 25! where the number of trailing zeros is known.</Outline>
</Outlines>
<Thread>
1: [Computes floor(42/5) = 8, floor(42/25) = 1, floor(42/125) = 0, then adds 8 + 1 + 0 = 9 and concludes that 42! has 9 trailing zeros, since 5s are the limiting factor compared to 2s.]
</Thread>
<Thread>
2: [Applies the same floor-sum to 10! and 25!, obtaining 2 and 6 trailing zeros respectively, which match the known values and confirm the correctness of the counting method.]
</Thread>
</Parallel>

<Parallel>
<Outlines>
<Outline>1: Reinterpret the floor-sum as directly counting multiples of 5 and extra factors from higher powers such as 25 among the integers from 1 to 42.</Outline>
<Outline>2: Optionally count powers of 2 in 42! to confirm that there are more than enough 2s so that 5s indeed determine the number of trailing zeros.</Outline>
</Outlines>
<Thread>
1: [Lists multiples of 5 up to 42: 5, 10, 15, 20, 25, 30, 35, 40 (8 numbers), each contributing at least one factor 5. Notes that 25 = 52 contributes one additional factor 5, giving 8 + 1 = 9 factors of 5 in total, consistent with the floor-sum calculation.]
</Thread>
<Thread>
2: [Computes the exponent of 2 in 42! as floor(42/2) + floor(42/4) + floor(42/8) + floor(42/16) + floor(42/32) = 21 + 10 + 5 + 2 + 1 = 39, which is much larger than 9, so the count of trailing zeros is indeed determined by the 9 factors of 5.]
</Thread>
</Parallel>

```

[Concludes that, since there are 9 factors of 5 and more than enough factors of 2, 42! has 9 trailing zeros.]

Final Answer
 $\boxed{9}$
</think>

To determine how many trailing zeros are in 42!, we look at how many times 10 divides 42!. Since $10 = 2 \cdot 5$ and factorials contain many more factors of 2 than 5, the number of trailing zeros is the exponent of 5 in the prime factorization of 42!. We use the standard formula for the exponent of a prime p in $n!$:

$$v_p(n!) = \sum_{k=1}^{\infty} \left\lfloor \frac{n}{p^k} \right\rfloor.$$

Here $p = 5$ and $n = 42$:

$$\left\lfloor \frac{42}{5} \right\rfloor = 8, \quad \left\lfloor \frac{42}{25} \right\rfloor = 1, \quad \left\lfloor \frac{42}{125} \right\rfloor = 0.$$

So the total number of factors of 5 in 42! is

$$8 + 1 + 0 = 9.$$

Since there are far more factors of 2 than 5 in 42!, each of these 9 factors of 5 pairs with a factor of 2 to form a factor of 10, which contributes one trailing zero.

Therefore, the number of trailing zeros in 42! is

9.

E Reasoning Trajectory Annotation Pipeline for ThreadWeaver

This section describes the multi-step pipeline that converts reasoning trajectories extracted from Qwen3-8B into a structured, parallelizable supervision signal for ThreadWeaver. Our pipeline consists of five steps:

1. Stage 1 uses GPT-5 to identify potential parallel structures within sequential reasoning traces.
2. Stage 2 rewrites these parallel structures into canonical <Thread> format.
3. Stage 3 performs revision against allowed and forbidden contexts to make the transition between context and each thread smooth.

4. Stage 4 regenerates an outline for every branch.
5. Stage 5 runs filtering to remove samples with failed rewriting or format errors to produce the final training corpus.

We describe the implementations of each step in detail.

Step 1 In Stage 1, raw trajectories are line-numbered and assigned stable identifiers (Lxx). This step ingests the resulting question–answer trajectories and has the language model reconstruct the entire chain of thought into a rigorously indexed outline that reuses these line identifiers. Each reasoning trace is presented in full, the major conceptual transitions are enumerated as top-level steps with precise line spans, and aborted explorations are retained to preserve the empirical distribution of strategies. The model is further instructed to decompose these top-level steps into contiguous substeps whenever they contain multiple thought units, producing a two-level hierarchy.

After the hierarchical labeling, the model analyzes dependencies to determine which steps or blocks can run concurrently, emitting sequential and parallel structures together with short explanations. Finally, the model serializes everything (step descriptions, substeps, and parallel threads) into a normalized XML-style summary. The line-range annotations introduced at the input are preserved in this summary and are propagated verbatim through all subsequent steps.

All the tasks in this step are conducted in a multi-round conversation, and the inputs and outputs from previous tasks are visible to later tasks.

Task 1: Main-step extraction and global indexing. Task 1 ingests the fully line-numbered reasoning transcript, where every sentence is prefixed with immutable tags such as L1, L2, and so on. Leveraging this dense context, the model enumerates all top-level steps sequentially, captures dead ends, and records contiguous line ranges Lxx-Lyy for each step, thereby establishing the canonical coordinate system that downstream tasks will reuse verbatim.

Task 1 Prompt for Step 1 Processing in ThreadWeaver

```
**Task 1: Main-Step Extraction**
Analyze the given long reasoning chain for a math problem and extract every *main* step of it. Ignore substeps—only list the top-level thinking processes or actions.

**Use the following output format:** 
```
S1: [Description of step 1] (Lxx-Lyy)
S2: [Description of step 2] (Lxx-Lyy)
...
SX: [Description of step X] (Lxx-Lyy)
```

**Guidelines:** 
- Label each top-level step consecutively (`S1`, `S2`, ...).
- Do *not* include any sub-numbering (no `S2.1`, etc.).
- Please capture the entire thought process presented in the reasoning chain, and do not skip any step that includes, but not limited to, the following:
  - Initial problem understanding and analysis
  - All exploration paths (both successful and unsuccessful)
  - Case studies, checks, or tests performed
  - Any “aha” or correction (re-evaluation or re-thinking) moments
  - The final reasoning that yields the solution
- Split multiple cases or scenarios into different steps. Each case should be allocated an independent step.
- If a reasoning path is started but later abandoned because it cannot reach a conclusion, still capture it as a distinct step.
- Please indicate the line numbers corresponding to where each main step starts and ends in the original reasoning chain (`Lxx-Lyy`).
  - Use the format `Lxx-Lyy` to specify the range of lines for each step.
  - S1 starts at the first line and ends at the line you list in the parentheses (`S1: [Description of step 1] (L1-Lyy)`).
```

- $S_{(k+1)}$ starts on the next line after S_k ends. The delta between the starting line number of $S_{(k+1)}$ and the ending line number of S_k is always 1.
- Ensure there is no gap between the end of one step and the start of the next.
- The last step must end at the final line of the original reasoning chain.
- Provide a brief summary of the content in each step that is one or two sentences long (e.g. `SX: [Description of step X]`).
- Keep the description for each step concise yet descriptive.
- A reasoning chain may contain any number of main steps. There is no limit to the total number of steps you can identify.

Task 2: Substep extraction within fixed ranges. Task 2 reuses both the raw transcript and the Task 1 step list as shared context, slicing each S_x span into adjacent substeps ($S_{x.1}$, $S_{x.2}$, ...) without gaps or overlaps. Because every line inside the parent range is still labelled with the original line numbers, the subdivision maintains identical Lxx-Lyy coverage while adding finer-grained anchors for later tasks.

Task 2 Prompt for Step 1 Processing in ThreadWeaver

Task 2: Substep Extraction

Given the previous output containing all main steps of a reasoning chain, break down each main step into substeps **only if** its content from the original reasoning chain can be meaningfully divided into smaller thought units. You can use the line ranges to reference the original reasoning chain and better capture the content of each main step.

Output format:

```
S1: [Description of step 1] (Lxx-Lyy)
  S1.1 [Description of substep 1.1] (Lxx-Lyy)
  S1.2 [Description of substep 1.2] (Lxx-Lyy)
  ...
  S1.X [Description of substep 1.X] (Lxx-Lyy)
S2: [Description of step 2] (Lxx-Lyy)
S3: [Description of step 3] (Lxx-Lyy)
S4: [Description of step 4] (Lxx-Lyy)
  S4.1 [Description of substep 4.1] (Lxx-Lyy)
  ...
  ...
SX: [Description of step X] (Lxx-Lyy)
  ...
  ...
```

Guidelines:

- A substep always uses the same parent index (`X`) as its main step (e.g. if breaking down `S2`, label `S2.1`, `S2.2`, ...).
- For each main step, capture the entire thought process presented in it, and do not skip any substep that includes, but not limited to, the following:
 - Initial problem understanding and analysis
 - All exploration paths (both successful and unsuccessful)
 - Case studies, checks, or tests performed
 - Any “aha” or correction (re-evaluation or re-thinking) moments
 - The final reasoning that yields the solution
- Split multiple cases or scenarios into different substeps. Each case should be allocated an independent substep.
- Provide a brief summary of the content in each substep (e.g., `S1.2 [Description of step 1.2]`) that is one or two sentences long.
- Keep each substep description concise and focused.
- Provide the line range for each substep (`Lxx-Lyy`).
 - The format Lxx-Lyy indicates the range of lines where the substep appears.
 - SX.1 starts at the first line of SX.
 - SX.(k+1) starts on the next line after SX.k ends.
 - Ensure there is no gap between the end of one substep and the start of the next.
 - The last substep must end at the final line of the parent main step.
- Keep the main step description and line ranges (`Lxx-Lyy`) as marked in Task 1.
- A main step may contain any number of substeps. There is no limit to the total number of substeps you can identify for a main step.
- If a main step does not contain substeps, copy over its step number, description and line range exactly as they appear in Task 1.
- Do *not* introduce nesting deeper than 2 levels. For example, `S2.1.1` is not allowed.

Task 3: Parallelizing main steps. Task 3 consumes the Task 1 summary and its line ranges, analyzes dependency relationships between adjacent steps, and expresses valid parallel groupings only when two steps do not rely on the same numbered lines. The reuse of the original annotations ensures that the resulting sequential and parallel blocks are traceable back to the same context without recomputing boundaries.

Task 3 Prompt for Step 1 Processing in ThreadWeaver

```
**Task 3: Parallelizing Main Steps**
Using only the **main steps** (`S1`, `S2`, ...) extracted in Task 1, first identify all steps or contiguous step groups that can be executed in parallel without violating logical dependencies; then rewrite them as a structured outline for parallel execution.

**Output format:**
```
Parallel groups:
P1: Parallel(S2, S3)
P2: Parallel(Sequential(S5, S6), S7)
...
PX: Parallel(Si, Sj, Sk...)

Outline for parallel execution:
S1: [Description of step 1] (Lxx-Lyy)
P1: Parallel[Parallel reason: ...](
 S2: [Description of step 2] (Lxx-Lyy),
 S3: [Description of step 3] (Lxx-Lyy)
)
S4: [Description of step 4] (Lxx-Lyy)
P2: Parallel[Parallel reason: ...](
 Sequential(
 S5: [Description of step 5] (Lxx-Lyy),
 S6: [Description of step 6] (Lxx-Lyy)
),
 S7: [Description of step 7] (Lxx-Lyy)
)
...
PX: Parallel[Parallel reason: ...](
 Si: [Description of step i] (Lxx-Lyy),
 ...
)
```

**Guidelines:**

1. **Identify Parallel Groups**


- Find groups of adjacent main steps with **no dependencies** among them.
- Label the groups as `P1`, `P2`, ... and represent the structure of steps within the group using the syntax pattern as shown above.
- Group steps Si, Sj, ... in `Parallel(Si, Sj, ...)` **if and only if** Sj does NOT depend on any part of Si.
- In a parallel block, you can combine adjacent steps that depend on each other into a `Sequential(Sx, Sy)` block. A sequential block can be executed in parallel to other main steps and sequential blocks.
- Do not create `Sequential` blocks that are not nested in any parallel block. Simply list the steps in order in those cases.
- Groups must cover contiguous step ranges in the original order.
- Interpretation of the problem should not be parallelized with the process of problem-solving.
- Different ways of self-reflection are often independent of each other and thus parallelizable.
- Self-reflection on an intermediate step is often parallelizable with the subsequent steps.
- In conditional logic, treat the **if** branch and **else** branch as independent tasks and parallelize them even though their outputs cannot both occur at runtime.
- Only include steps that can be executed in parallel with others in a parallel group; the parallel groups do not need to cover every step.
- A parallel group should not consist of only one single step or one `Sequential(...)` block. If this is the case, do not include those steps into any parallel groups.



2. **Write the Structured Outline for Parallel Execution**


- Annotate the main steps with the parallel grouping structures.
- Keep each main step's description and line range the same as those annotated in Task 1 (preserve as much as you can).
- Include **every** step exactly once, either alone, inside a sequential block, or inside a parallel group.

```

- If a main step does not belong to any sequential block or parallel group, copy over its step number, description and line range exactly as they appear in Task 1. Include it in the outline in its original position.
- For a sequential block, list each step in order.
- If a parallel group contains multiple units, briefly state why its units are independent in the `parallel reason` bracket (e.g. `PX: Parallel[Parallel reason: ...]`).
- **Contiguous Blocks:** You may combine multiple adjacent steps (e.g. `Sequential(S1, S2, ...)`) and treat them as a single unit in the parallel group alongside another step or block (e.g. `S3`), but do **not** combine non-adjacent steps into a block.
- **Contiguous grouping:** Each parallel group must cover a continuous sequence of steps or blocks. In other words, you may only parallelize adjacent steps. The occurrence of steps in parallel groups must follow their original order. For example, P1: Parallel(S5, S8) is not allowed.
- **Strict Parallelism Only:** Build an explicit dependency graph in your analysis: draw an edge from step A to step B if B uses A's output or insight. A group `Pi` may include steps (or contiguous blocks) **only** if there are **no** dependency edges between any two units.

Task 4: Parallelizing substeps. Task 4 layers on top of the substep catalog from Task 2 and the parallel structure from Task 3, examining each main step in isolation to see which adjacent substeps can proceed concurrently. Because the inputs still reference the exact Lxx-Lyy intervals, the model can safely bundle parallel-friendly fragments while citing the same line numbers, preserving context reuse down to the second hierarchy level.

Task 4 Prompt for Step 1 Processing in ThreadWeaver

```
**Task 4: Parallelizing Substeps**
Examining the outline you extracted in Task 3, for any main steps that are not in a `Parallel` block, refer to its
**substeps** you extracted in Task 2 if there is any (e.g. `S1.1`, `S1.2`, ...). Among these substeps, identify all
substeps or contiguous substep groups that can be executed in parallel without violating logical dependencies. Then
rewrite the structured parallel execution outline to include both the parallel structures of the main steps and those of
the substeps you just identified.

**Output format:**
```
Parallel groups of substeps:
* Parallel(Sequential(S1.1, S1.2), S1.3)
* Parallel(S4.2, S4.3)
* ...
* Parallel(...)

Outline for parallel execution:
S1: [Description of step 1] (Lxx-Lyy)
P-sub: Parallel[Parallel reason: ...](
 Sequential(
 S1.1: [Description of substep 1.1] (Lxx-Lyy),
 S1.2: [Description of substep 1.2] (Lxx-Lyy)
),
 S1.3: [Description of substep 1.3] (Lxx-Lyy)
)
P1: Parallel[Parallel reason: ...](
 S2: [Description of step 2] (Lxx-Lyy),
 S3: [Description of step 3] (Lxx-Lyy)
)
S4: [Description of step 4] (Lxx-Lyy)
S4.1: [Description of substep 4.1] (Lxx-Lyy)
P-sub: Parallel[Parallel reason: ...](
 S4.2: [Description of substep 4.2] (Lxx-Lyy),
 S4.3: [Description of substep 4.3] (Lxx-Lyy)
)
P2: Parallel[Parallel reason: ...](
 Sequential(
 S5: [Description of step 5] (Lxx-Lyy),
 S6: [Description of step 6] (Lxx-Lyy)
),
 S7: [Description of step 7] (Lxx-Lyy)
)
...
PX: Parallel[Parallel reason: ...](
```

```

Si: [Description of step i] (Lxx-Lyy),
...
)
```

**Guidelines:**
1. **Identify Parallel Groups**

- Only parallelize the substeps whose main steps are not in a parallel block.
- Find sets of adjacent substeps with no dependencies among them.
- Label the groups of substeps `P-sub`, and list their substep members (e.g. `Parallel(Sequential(S1.1, S1.2), S1.3)`, `Parallel(S4.2, S4.3)`).


2. **Write the Structured Outline for Parallel Execution**

- Annotate the main steps and substeps with the parallel group structures.
- Keep the parallel group structure, numbering, description and line ranges of the main steps the same as those annotated in Task 3 (preserve as much as you can).
- Keep each substep's numbering, description and line range the same as those annotated in Task 2 (preserve as much as you can).
- For a sequential block, list the description of each substep in order.
- If a parallel group contains multiple units, briefly state why its units are independent in the `parallel reason` bracket (e.g. `PX: Parallel[parallel reason: ...]`)
- If a substep does not belong to any parallel or sequential group, copy over its step number, description and line range exactly as they appear in Task 2. Include it in the outline in its original position.



- **Coverage:** For each main step you choose to annotate, include **every** substep exactly once, either alone or inside a parallel group.



- **Contiguous Blocks:** In a parallel block, you may combine adjacent substeps that depend on each other into a contiguous block (e.g. `Sequential(S2.1, S2.2)` and treat them as a single unit in the parallel group alongside another substep or block (e.g. `S2.3`). Do **not** combine non-adjacent substeps into a block.
- Do not create `Sequential` blocks that are not nested in any parallel block. Simply list the substeps in order in those cases.
- Only group substeps corresponding to the same main step in a single-level parallel or sequential block (e.g. Parallel(S2.2, S3.1) or Sequential(S2.1, S2.2, S3.1) is not allowed).
- **Strict Parallelism Only:** Build an explicit dependency graph in your analysis: draw an edge from substep A to substep B if B uses A's output or insight. A group `Pi` may include steps (or contiguous blocks) **only** if there are **no** dependency edges between any two units.
- In conditional logic, treat the **if** branch and **else** branch as independent tasks and parallelize them even though their outputs cannot both occur at runtime.
- Contiguous grouping only: Each parallel group must cover a continuous sequence of steps or blocks. In other words, you may only parallelize adjacent steps or substeps. The occurrence of steps or substeps in parallel groups must follow their original order. For example, P1: Parallel(S2.2, S3.1) is not allowed.

```

Task 5: XML-style serialization with ranges. Task 5 repackages everything derived so far—main steps, substeps, and parallel groupings—into the prescribed XML-like outline. Each `<Thread>` and `<Parallel>` element explicitly repeats the inherited Lxx-Lyy spans, so the serialized structure is a faithful rendering of the earlier outputs that downstream tooling can align with either the raw transcript or the intermediate tables.

Task 5 Prompt for Step 1 Processing in ThreadWeaver

****Task 5: Annotate the Overall Structured Outline with Line Range for Each Parallel Block****
According to the conversation above, output the overall parallel execution structure of the long reasoning chain with the main steps, substeps and the parallel structures clearly defined in a simple XML format.

In addition, include the line ranges for each parallel block and its threads.

****Output format:****
```
S1: [Description of step 1]
<Parallel>[Parallel reason: ...]
<Thread>
S1.1: [Description of substep 1.1]
S1.2: [Description of substep 1.2]
</Thread> (range: Lxx-Lyy)
<Thread>
S1.3: [Description of substep 1.3]
</Thread> (range: Lxx-Lyy)
```

```

</Parallel> (range: Lxx-Lyy)
<Parallel> [Parallel reason: ...]
<Thread>
S2: [Description of step 2]
</Thread> (range: Lxx-Lyy)
<Thread>
S3: [Description of step 3]
</Thread> (range: Lxx-Lyy)
</Parallel> (range: Lxx-Lyy)
S4: [Description of step 4]
  S4.1: [Description of substep S4.1]
  <Parallel> [Parallel reason: ...]
  ...
  </Parallel> (range: Lxx-Lyy)
<Parallel> [Parallel reason: ...]
<Thread>
S5: [Description of step 5]
S6: [Description of step 6]
</Thread> (range: Lxx-Lyy)
...
</Parallel> (range: Lxx-Lyy)
...
```
Guidelines:
```

- Format the output \*\*strictly\*\* according to the output format specified above. \*\*The only XML tags allowed are `<Parallel>` and `<Thread>`.\*\* A `<Parallel> </Parallel>` block should consist of multiple `<Thread>`'s. `<Thread>` can only exist inside `<Parallel>`.
- A `<Parallel>` block may form at either the main step level, or the substep level.
- Ensure \*\*there are no `<Parallel>` tags nested inside one another.\*\* Do not create structures like `<Parallel> ... <Parallel> ... </Parallel> ... </Parallel>`.
- \*\*Max depth of substep nesting is 2.\*\* Ensure there is no step number beyond `Sx` and `Sx.y`. For example, ``Sx.y.z`` is not allowed.
- \*\*Do not add additional tags for sequential structures.\*\* If an XML node's children are purely sequential, list them in order.
- \*\*Tag parallel blocks.\*\* Wrap parallelizable units in ``<Parallel>` ... `</Parallel>``. Wrap each parallelizable unit (including sequential blocks and (sub)steps) in ``<Thread>` ... `</Thread>``. List the sequential steps and substeps in order inside each `<Thread>` block.
- Provide the line range for each `<Thread>` block after the closing tag (``</Thread> (range: Lxx-Lyy)``).
- Provide the line range for each `<Parallel>` block after the closing tag (``</Parallel> (range: Lxx-Lyy)``).
- Keep the description for every step and substep as annotated in Task 1 and 2 (preserve as much as you can).
- \*\*Do not create significantly imbalanced parallel blocks.\*\* If most of an XML node's children are sequential and only a small number of them can run in parallel, you may leave the group untagged or only create a parallel block using the children that are genuinely parallel.
- Do not start your output with "```xml". Indentation (i.e., spaces before `Sx.y`) indicates substeps, not XML structures; do not add tags other than `<Parallel>` and `<Thread>` around them.
- \*\*Group parallelizable sets.\*\* You may bundle several independent units into a single ``<Parallel>`` block when they share no dependencies.

**Task 6: Consistency verification across contexts.** Task 6 revisits the complete context (the original transcript plus all prior task outputs) to double-check that every instruction was satisfied. By reusing the same line numbering and dependency notes, it validates that no steps were skipped, that parallel reasoning is justified, and that references respect the allowed-versus-forbidden context policy introduced earlier in the prompt.

#### Task 6 Prompt for Step 1 Processing in ThreadWeaver

**\*\*Task 6: Double-check Every Parallel Block and Refine the Structured Outline\*\***  
 Please check and revise the structured outline for parallel execution you generated in Task 5 by ensuring the following:

- **\*\*Coverage & Contiguity\*\*:** Make sure the line range (Lxx-Lyy) of each thread exactly spans from the first included (sub)step to the last. Make sure the line range (Lxx-Lyy) of each parallelizable block exactly spans from the first included thread to the last.
- **\*\*Parallelism Validity\*\*:** Verify that there are no direct or indirect dependencies among the children of each parallel block. Specifically, confirm that each thread in the block does not depend on \*any\* of the threads before it in the same block. Add a note after the closing tag of each thread (``</Thread>``) indicating the result of this check.
  - If the check passes, add the note ``check passed`` and confirm the reason (``</Thread> [range: Lxx-Lyy, check passed: reason]``).

- If the check fails (i.e. dependency is detected within the parallel block), refactor the problematic block into Sequential(...) or separate groups, and update the line ranges accordingly. If you cannot determine how to refactor, leave a note stating `check failed` along with the reason (`</Thread> [range: Lxx-Lyy, check failed: reason]`).
- \*\*Output Format\*\*: Keep the outline format as specified in Task 5. Only the tags `<Parallel>` and `<Thread>` are allowed, and `<Thread>` tags can only appear inside `<Parallel>` blocks.

```
Output format:

```
S1: [Description of step 1]
<Parallel>[Parallel reason: ...]
<Thread>
S1.1: [Description of substep 1.1]
S1.2: [Description of substep 1.2]
</Thread> (range: Lxx-Lyy, check passed: first thread)
<Thread>
S1.3: [Description of substep 1.3]
</Thread> (range: Lxx-Lyy, check passed: S1.3 has no dependencies on S1.1 and S1.2)
</Parallel> (range: Lxx-Lyy)
<Parallel>[Parallel reason: ...]
<Thread>
S2: [Description of step 2]
</Thread> (range: Lxx-Lyy, check passed: first thread)
<Thread>
S3: [Description of step 3]
</Thread> (range: Lxx-Lyy, check failed: S3 depends on S2)
</Parallel> (range: Lxx-Lyy)
S4: [Description of step 4]
S4.1: [Description of substep S4.1]
<Parallel> [Parallel reason: ...]
...
</Parallel> (range: Lxx-Lyy)
<Parallel> [Parallel reason: ...]
<Thread>
S5: [Description of step 5]
S6: [Description of step 6]
</Thread> (range: Lxx-Lyy, check passed: first thread)
...
</Parallel> (range: Lxx-Lyy)
...
```

```

*Step 2* The second step is a deterministic restructuring pass that converts the free-form sequences emitted above into canonical parallel markup. It sweeps over every annotated trajectory, infers the boundaries of each reasoning branch, and wraps the contiguous spans inside explicit thread containers that sit beneath the appropriate parallel block. During this pass the tool enforces ordering constraints, ensures that adjacent steps remain contiguous after threading, and drops in the dependency notes that certify why two segments may or may not execute simultaneously. Crucially, the script preserves the inline Lxx prefixes that Step 1 injected into every reasoning line, so the downstream prompts can continue to reference exactly the same line-number ranges even after the text is reorganized into <Thread> blocks. Because the logic consists solely of programmatic rewrites plus structural validation, no additional prompting is necessary.

*Step 3* Next, each candidate trajectory undergoes a linguistic compliance review that compares the text under inspection against immediate predecessors and lateral sections. The reviewer model receives a slice of “allowed” context plus a sibling “forbidden” context, scans the target passage for implicit carry-overs, and proposes fine-grained edits to excise any leakage from the forbidden branch. The output takes the form of minimal substring replacements in JSON, which allows automated application while keeping every other token untouched. Running this step across all paths eliminates hidden cross-references, making each branch self-contained before outline regeneration.

## Prompt for Step 3 Processing in ThreadWeaver

In this task, your goal is to check the English language of a given text segment against two other text segments from the same article:

- \* The "allowed context" segment comes from the immediate preceding section of the text segment under review. The text segment under review may reference content from this previous section.
- \* The "forbidden context" segment is from a section \*\*parallel\*\* to the text segment under review. The text segment under review must not reference any content from this parallel section.

Specifically, you will be provided with the following:

1. Allowed Context — contains information the text under review may reference.
2. Forbidden Context — contains information the text under review must not reference.
3. Text Under Review — the text segment to be checked.

Your task is to decide whether the text under review refers to any information in the Forbidden Context.

If it does, output a JSON array of substring replacements that remove or rewrite the referencing language in the text under review, following the output format specified below.

If it does not, output an empty array: `[]`.

Output the raw JSON only. Do not enclose the output in "```json...```".

Output format:

```
```
[
  {
    "src": "<the substring to be removed or rewritten (exact match, properly escaped)>",
    "dest": "<the replacement text which removes the reference to the Forbidden Context (use \"\\\" to delete)>"
  },
  ...
]
```

Guidelines:

- * Replace the minimal substrings needed to eliminate the reference to the Forbidden Context while keeping the text under review grammatical.
- * Escape quotes and newlines in JSON (use `\"` and `\\n`).
- * A proposed replacement will affect all occurrences of the "src" field in the text under review. If you do not intend to modify all the occurrences, make the substring in "src" more specific.
- * Do not introduce new information from the Forbidden Context in your replacements.
- * References to the Allowed Context are permitted. Do not remove any references to the Allowed Context.
- * Do not add explanations or code fences.
- * Do not output any text outside the JSON.

Special cases to watch for:

- * Remove or rewrite any occurrences of:
 - Anaphora/Deictics: Phrases like "as above," "as before," "same as earlier," "that result," "those values," which implicitly point to the Forbidden Context. Keep these when they refer to the Allowed Context.
 - Implicit carry-over: "unchanged from the previous section," "as we computed," when the "previous" information is from the Forbidden Context.
 - Shorthand references: Symbols, variable names, or equation labels defined only in the Forbidden Context (e.g., "that z," "the value from (3)").
 - Cross-references: "see previous," "see above," footnote markers, or inline citations that point to the Forbidden Context.
 - Paraphrased pointers: Rewrites that still rely on the Forbidden Context even without verbatim overlap.
- * Overlapping matches: When multiple candidate substrings overlap, prefer one replacement of the longest meaningful phrase rather than many small ones.
- * Case/whitespace variants: Handle differences in capitalization or spacing exactly by matching the literal substring present in the path.
- * Multiline fragments: If the referencing text to be replaced spans lines, include `\\n` in src and dest as needed.
- * Grammar preservation: If removing a piece of referencing text leaves dangling punctuation or grammar issues, include the surrounding words/punctuation in "src" to keep the sentence clean.
- * Nothing salvageable: If an entire clause or sentence is a forbidden reference (e.g., "Which is the same as before."), delete it by setting "dest": "".

Allowed Context:

```
...
[5 lines before the <Parallel> block]
```

Forbidden Context:

```
[First 5 lines at the beginning of the prior <Thread> block]
```

...
[Last 5 lines at the end of the prior <Thread> block]

Text Under Review:
[Full text of the <Thread> block to check]

Step 4 Once the sanitized paths are in place, a fresh model call rebuilds the planning scaffolds that will ultimately supervise downstream learners. Using only the contextual window that precedes each parallel block, the model drafts concise yet actionable outlines for every branch, restating the tactical moves a mathematician would plan before executing the proof. Each outline mirrors the ordering of the paths, isolates the reasoning intent without revealing conclusions, and keeps the different branches independent so later steps can treat them as interchangeable schedules. Because the outlines are derived strictly from the provided context, they serve as high-quality teacher plans for training planning modules or evaluating interpretability.

Prompt for Step 4 Processing in ThreadWeaver

In this task, your goal is to generate outlines for some mathematical reasoning procedures. Specifically, you will be given

1. A coherent math reasoning trace segment tagged with parallel structures (`<Parallel> ... <Thread> ... </Thread> ... </Parallel>`) that indicate reasoning steps which can be executed in parallel.
2. A group of placeholders placed at the beginning of the segment
```

```
<Outlines>
<Outline>1: [Outline to be generated for path 1 below]</Outline>
<Outline>2: [Outline to be generated for path 2 below]</Outline>
...
<Outline>X: [Outline to be generated for path X below]</Outline>
<Outlines>
```

3. A segment of text immediately preceding the target reasoning trace segment ('`Context:`') to serve as the context for deriving the outline.
```

Using only the provided context ('`Context:`'), generate the outline for each <Thread> in the <Parallel> segment. Place the generated outline in the markers "[Outline to be generated for path X below]".

Output format:

```
```
```

```
<Outlines>
<Outline>1: [Outline to be generated for path 1 below]</Outline>
<Outline>2: [Outline to be generated for path 2 below]</Outline>
...
<Outline>X: [Outline to be generated for path X below]</Outline>
<Outlines>
```
```

Guidelines:

- * Imagine that a mathematician is thinking and using the outlines to structure and guide their thinking process in each corresponding path ('`<Outline>X:... <Thread>X:...`').
- * Frame the outlines so the mathematician could logically plan them based solely on the context, without access to the content in the parallel segment or any additional information.
- * Make the outline directly and easily derivable from the context – neither too abstract nor too specific – and strictly aligned with the order of the paths.
- * Each outline should capture the gists and logical structure of its path without excessive details.
- * Thinking in each path is done independently. Ensure that the outlines remain distinct and do not depend on one another.
- * Do not reveal exact calculations, numerical results, or the final conclusion from its path in the outline. Instead, indicate the steps/methods/directions that lead to them.
- * Do not introduce a substantial amount of reasoning in the outlines.
- * Avoid phrases like "conclude that ...", "rule out that ...", or anything that cannot be directly derived from the context without significant reasoning. Do not frame the outline as if the conclusion is already known.
- * A reader should be able to write the path from the outline plus the context alone with some reasoning effort, without additional information.
- * Keep the outline concise and clear: one sentence for short content; at most two sentences for longer content.
- * Output plain text within each outline block: no bullet points, numbering, headings, or extra commentary.

* Do not start your output with "```xml". Start your output with <Outlines> and end your output with </Outlines>. The only allowed tags in a <Outlines> block are <Outline> tags.

Step 5 The final step is a comprehensive filtering pass that operates on the model-generated artifacts. It traverses every outline file, enforces formatting contracts, checks that parallel tags are balanced, and discards trajectories that still violate structural constraints after the preceding corrections, ensuring that only high-quality, well-structured demonstrations enter the supervised learning pipeline.