Instilling Parallel Reasoning into Language Models

Matthew Macfarlane[*1] **Minseon Kim**[2] **Nebojsa Jojic**[2] **Weijia Xu**[2] **Lucas Caccia**[2] **Xingdi Yuan**[2]
Wanru Zhao[*3] **Zhengyan Shi**[2] **Alessandro Sordoni**[2]

Abstract

Sequential chain-of-thought reasoning significantly improves the performance of Large language models (LLMs) on complex tasks. However, sequential reasoning has structural limitations: Long chains are expensive due to attention's quadratic complexity, and multiple diverse strategies cannot be considered simultaneously. To address this we propose a method that instills parallel reasoning capabilities in LLMs by distilling parallel reasoning traces from a teacher model. This approach enables models to decompose problems, explore diverse strategies via concurrent reasoning traces, and aggregate trace outputs for the final answer. Evaluating on a variety of math and puzzle benchmarks such as MATH 500, AIME and Countdown, we show our approach can decompose parallelizable problems, and that the performance scales with the number of parallel traces. The resulting model can dynamically allocate reasoning strategies based on problem complexity, outperforming standard sampling methods.

1. Introduction

Large Language Models (LLMs) have demonstrated remarkable improvements on benchmarks that benefit from intensive reasoning, largely driven by prompting methods that scale test-time compute through extensive generation, such as Chain-of-Thought (CoT) prompting (Wei et al., 2022). CoT encourages step-by-step reasoning, allowing models to explicitly generate intermediate steps instead of relying on implicit computations within their activations.

Recent research (Shao et al., 2024) has aimed to reinforce and refine the sequential reasoning behaviors during pre-training. While step-by-step reasoning has been effective in improving performance, sequential reasoning comes with structural limitations for efficient problem solving. Scaling to larger sequence lengths incurs significant latency and computational overhead, which scales quadratically w.r.t the input length for exact attention. Second, errors made early in the sequence can compound throughout the reasoning process (Wu et al., 2025), leading to unstable or incorrect final answers. To address these limitations, we propose a parallel reasoning approach that enables models to explore multiple reasoning paths simultaneously, improving both efficiency and robustness.

Parallel computation methods in LLM inference, such as majority voting (Wang et al., 2022), and self-certainty (Kang et al., 2024), have been used to improve inference-time performance by aggregating multiple independent reasoning traces. However, since these traces are generated independently from the same prompt and rely solely on sampling, they often result in similar or identical outputs. This limits the diversity and effectiveness of parallel inference, highlighting the need for a more structured and coordinated approach to fully exploit the potential of parallel reasoning.

Our approach builds a parallel reasoning strategy directly into the chain of thought, where threads collaborate to explore diverse reasoning traces or collectively breakdown inherently parallelizable computations. This strategy prevents redundant computations through coordination among the parallel threads before execution. However, such parallel reasoning strategies cannot emerge from current training regimes, which are designed for sequential auto-regressive generation. However we find that teacher models can be used to to simulate parallel reasoning, which when combined with a verification step provides high quality targets, which can then be used to teach a student model to execute true parallel reasoning.

We explore explicitly instilling the ability to perform parallel reasoning into LLMs. Our central insight is that we can use a highly capable "teacher" model to generate high-quality, parallelizable reasoning traces but generated via a sequential generation process. This data can then be used to train a model to execute the inference in parallel.

We show the following. First, we effectively distill the abil-

[1]University of Amsterdam *Work done while at Microsoft Research, Montreal. [2]Microsoft Research, Montreal, Canada [3]University of Cambridge. Correspondence to: Matthew Macfarlane <matthew.v.m@live.co.uk>.

2nd AI for Math Workshop @ 42^{nd} International Conference on Machine Learning, Vancouver, Canada, 2025.

ity to decompose inherently parallelizable problems and generalize to decomposable tasks beyond the training distribution into a student model. Second, we demonstrate that scaling the number of parallel reasoning threads serves as a powerful scaling axis for improving the performance. Lastly, we show that the trained model is able to perform efficient inference while maintaining high performance, leveraging the benefits of parallelism, compared to chain of thought budget-forcing methods.

2. Related Work

Test-time inference strategies aim to improve LLM performance without modifying model weights, instead utilizing different strategies for decoding the auto-regressive model during inference. These can be broadly divided into sequential and parallel approaches. Chain of thought prompting (Wei et al., 2022) introduced auto-regressive generation of intermediate tokens before generating the answer, encouraging the model to think through the solution before arriving at an answer. Numerous additional iterations of this inference strategy have been proposed, such as Tree of Thoughts (ToT) (Yao et al., 2023) allowing for look-ahead and backtracking and Graph of Thoughts (GoT) (Besta et al., 2023) which models reasoning as a directed acyclic graph.

2.1 Parallel Inference

Sampling-based verifier-free methods such as self-consistency (Wang et al., 2022) are a form of parallel reasoning, generating multiple samples from a model and selecting the most consistent answer through voting, mitigating erroneous steps, especially in arithmetic and logical tasks. Kang et al. (2024) extend this with a self-certainty-based best-of-N selection, using output distribution peakedness to guide weighted voting aggregation. Verification-based filtering, such as with generative verifiers (Zhang et al., 2024), use a LLM to filter incorrect paths, combining parallel exploration with robust evaluation. Probabilistic inference methods that leverage parallelism have also been explored; Zhao et al. (2024) use Sequential Monte Carlo (Liu & Chen, 1998), viewing LLM inference as probabilistic sampling from an unnormalized distribution. Parallel tree search methods like Dynamic Parallel Tree Search (DPTS) (Ding et al., 2025) accelerate reasoning by exploring diverse tree nodes concurrently, dynamically prioritizing promising paths for faster, accurate problem-solving. Fractured Chain-of-Thought (Liao et al., 2025) explores the trade-off between scaling compute in sequence length and parallel samples. Scaling inference compute along the parallel dimension has proven an effective way to improve decision making. The "Large Language Monkeys" framework (Brown et al., 2024) showed massive parallel sampling significantly boosts correctness (e.g., in code generation). Inference scaling laws (Wu et al., 2024; AbdElhameed & Halim, 2024) further investigate compute-optimal strategies, often favoring parallel compute over model size scaling.

2.2 Training for Parallel Inference

While it is possible to leverage parallel inference in existing LLMs without any explicit training, various works have investigated the benefits from explicitly training for this type of test time inference. PASTA (Jin et al., 2025) trains models to emit semantically disjoint output segments for asynchronous decoding, reducing latency. Skeleton-of-Thought (SoT) (Ning et al., 2023) prompts models for an initial concise skeleton, then expands points in parallel, speeding up structured responses, with an adaptive router (SoT-R) for selective application. Hogwild! Inference (Rodionov et al., 2025) enables parallel token generation via multiple workers with a shared Key-Value cache, allowing existing LLMs to collaborate dynamically without fine-tuning. APR (Pan et al., 2025) extends Stream of Search (Gandhi et al., 2024) by generating training data with explicit heuristic examples in the dataset of search strategies for components that could be executed in parallel and then fine tune this policy with GRPO (Shao et al., 2024). The Parallel Scaling Law for Language Models (Chen et al., 2025b) explores using parallel inference to improve each next token prediction by generating P independent threads that have different prefix embeddings but share the same parameters for each token generation. Multiverse (Yang et al., 2025), similar to our approach, use a teacher model, however instead of generating data explicitly for parallel reasoning they use existing chain of thought data and find ways to parallelize it. This reduces the cost of generation but means the data does is not designed to leverage parallel reasoning as much as possible leading to low utilization in practice.

Reasoning in continuous spaces is another promising direction for being able to reason in parallel at test time. Coconut (Hao et al., 2024) performs reasoning in a continuous latent space ("latent chain-of-thought"), potentially allowing implicit parallelism, but does not demonstrate that this type of parallel reasoning indeed emerges. Lastly, architectures like Universal Transformers (Dehghani et al., 2018), with iterative layer application at test time, could theoretically perform parallel reasoning, though current methods might not explicitly learn these strategies.

3. Background

3.1 Chain of Thought

Chain-of-thought (CoT) prompting (Wei et al., 2022) encourages large language models (LLMs) to generate intermediate reasoning steps before producing a final answer. This approach leverages the autoregressive nature of LLMs, where each token is predicted based on prior context. Including reasoning traces in the prompt increases the likelihood of accurate responses, especially for tasks requiring logical or multi-step reasoning. For a question q, the probability of generating answer a, denoted $p(a \mid q)$, is improved by conditioning on a sequence of reasoning steps r, yielding $p(a \mid q, r)$, where r represents intermediate tokens mimicking human-like reasoning.

3.2 Budget Forcing

Chain-of-thought prompting allows the model to decide when to stop reasoning and generate an answer, which can lead to under-thinking (insufficient reasoning) or overthinking (excessive reasoning that introduces errors). Budget forcing is a test-time prompting strategy that enforces a fixed reasoning budget B, measured in tokens, before an answer is produced. If the model tries to end reasoning early (before B tokens), continuation prompts like "wait, think some more" are appended to extend the process. Once B tokens are reached, reasoning is terminated, often with a token like `</think>`.

4. Methodology

Our approach enhances reasoning in LLMs by introducing two complementary and parallelizable reasoning strategies: diversify and decompose. The diversify strategy generates multiple instructions that explore different solution strategies for a given problem. The decompose strategy takes each instruction and breaks it down into smaller sub-problems, i.e., threads, that can be addressed independently. Both strategies are learnable operations that enable the exploration of multiple reasoning paths and the division of complex problems into sub-problems. We believe these two core skills can be a foundation for inherently parallel reasoning, where on-policy learning could be leveraged to generate policy improvement.

In the following sections, we describe our core contribution: a method to train a student model to learn both the diversify and decompose reasoning strategies. Specifically, in Section 4.2, we describe the process of generating synthetic training data using a teacher model that simulates parallel reasoning strategies. Section 4.3 details how this data is used to train a student model to imitate these strategies. Finally, in Section 4.4, we demonstrate how the trained student model executes parallel reasoning at inference time. We highlight that we are able to teach these parallelism strategies using a teacher model, which does not have ability to execute parallelism strategies, but can instead simulate such strategies via a sequentially generated trace that mimic parallelism. This distinguishes our work from previous approaches that attempt to identify parallelism opportunities in existing chain of thought traces.

4.1 Parallel Reasoning Strategies

Diversify reasoning involves exploring multiple, fundamentally different approaches to solve a problem. For instance, when evaluating a chess position, the model can generate several hypotheses for the best move and investigate the consequences of each in parallel. Such parallelism is distinct from naive sampling as it explicitly coordinates the reasoning paths to pursue diverse strategies, thereby preventing redundant computations. Moreover, it also allows the exploration of riskier strategies that might yield high rewards but would normally have low likelihood under a prior policy optimized for single-sample performance.

Decompose reasoning focuses on breaking down a single complex strategy for solving a problem into smaller, self-contained sub-problems. These sub-problems can be computed independently and in parallel before their results are aggregated to form the final solution. For example, a complex mathematical calculation can be solved by concurrently evaluating its independent sub-expressions and then combining the results. This type of parallelism is primarily aimed at improving inference speed by leveraging computations that can be executed in parallel.

Prompt template

```
Parallel Reasoning Mode:

I will use N threads for this

<thread 1> Instruction 1 </thread 1>
<thread 2> Instruction 2 </thread 2>
<thread 3> Instruction 3 </thread 3>

<thread 1> Response 1 </thread 1>
<thread 2> Response 2 </thread 2>
<thread 3> Response 3 </thread 3>

<think> Summary </think>
<answer> Final Answer </answer>
```

4.2 Data Generation

Our data generation process is designed to create structured data points for training. Each data point is a complete collection represented by the tuple (x, N, I, R, c, a), where x is the problem statement, N is the number of parallel thread instruction, $I = \{instr_1, instr_2, \cdots, instr_N\}$ is the set of thread instructions, $R = \{r_1, r_2, \cdots, r_N\}$ is the set of corresponding thread responses, and c and a are the final summary and answer, respectively.

For both diversify and decompose strategies, we generate this data sequentially using a single chain-of-thought process from a teacher model. The key is a prompt-and-verify

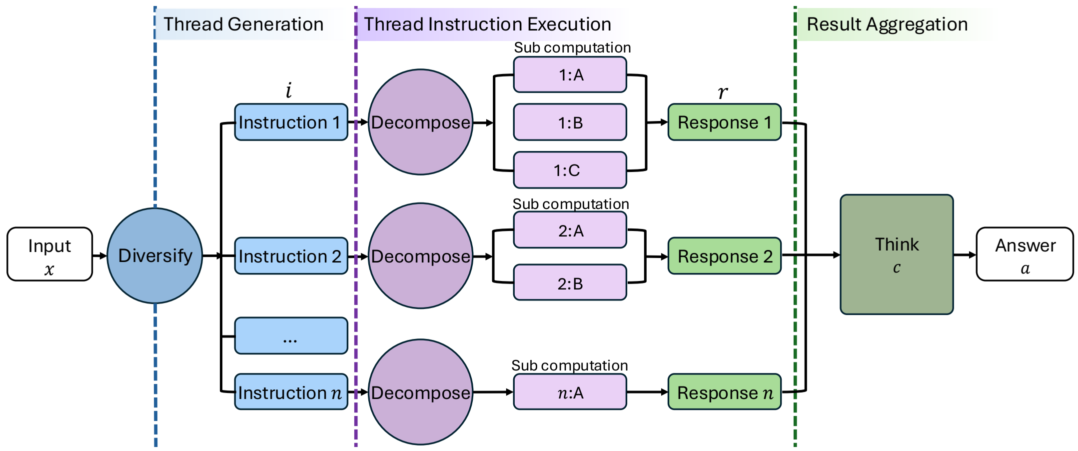

Figure 1: Inference using both the Diversify and Decompose parallel reasoning strategies. Diversify is first applied to try to solve the problem using many diverse strategies; Decompose is then used to break down these targeted approaches into parallelizable computations.

method that ensures the resulting threads are sufficiently distinct and independent for parallelization, as assessed by an LLM-as-a-Judge (Zheng et al., 2023). We prompt the teacher model to generate the components of the tuple in a specific structured format, as illustrated below.

Grok-3-mini (xAI, n.d.) was used for data generation, due being the only model evaluated (also investigated Gemini-2.5.flash, GPT 4.0 mini) that could consistently produce data in this structured format. Adherence to this format was a higher priority than achieving policy improvement during the Supervised Fine-Tuning (SFT) process itself. Hence Grok-3-mini was chosen over other models that have top performance on standard benchmarks.

4.2.1 Diversify Data Generation

For generating data to learn the diverse reasoning strategy, the teacher model was prompted to generate N threads, each representing a different approach to a given problem. To ensure the quality of this diversity, we employ an LLM-based judge to filter out any generated samples where the threads are not sufficiently distinct from one another.

A key consideration is determining the appropriate number of threads, N, for any given problem. Simpler problems may require only a single thread, whereas more complex problems can benefit from a multitude of approaches. To formalize this, we sample from the distribution $p(n_{\text{min}}|x)$, which represents given a single sample what is the minimum number of threads the teacher model requires to solve a problem x. We sampled from this distribution using the following iterative data collection method: For each problem, we begin by attempting a solution with a single thread ($N = 1$). If this attempt is successful, the data point is added to our dataset. If it fails, we double the thread count and generate a new sample. This process is repeated up to a maximum of 64 threads. If the problem remains unsolved after reaching this limit, it is discarded.

This iterative approach serves two critical functions. First, it encourages a wide variety of thread counts within the dataset, addressing the tendency of the model to otherwise lack diversity in the number of threads it self-selects. Second, it naturally allocates more computational resources—in the form of higher thread counts—to more difficult problems, precisely where they are most needed.

4.2.2 Decompose Data Generation

For decomposition, we prompt the model to consider a problem and a specific solution strategy, and then to break down that strategy into parallelizable sub-tasks. We leverage two types of data for decomposition generation. Synthetic data where the number of possible threads the task can be decomposed into is known and forced in prompting. We also use self generated data using the task, problem pairs from the diversify generation phase. In this case, we allow the teacher model to determine the appropriate number of threads since this value is not a priori known.

For all data points we perform an independence check using LLM-as-a-judge to check that the computations of each thread do not rely on each other. We perform a maximum of 3 retries to generate data with independent threads, after which the data-point is discarded. See Appendix B for full details including prompts.

4.3 Training

In order to enable parallelism at inference, we train a single model with parameters θ, to perform three distinct types of inference, each guided by a unique prompt.

The first type is **Thread Instruction Generation**. The model learns a single function conditioned on the problem x and the prompt p_i (note this is different depending on whether we are performing diversify or decompose reasoning). In order to enable a thread count chosen by the model or by the user, we factorize instruction generation by first generating the thread count and then the thread instructions:

$$p_\theta(n, instr \mid p_i, x) = p_\theta(n \mid p_i, x) \cdot p_\theta(instr \mid p_i, x, n)$$

The second type is **Thread Instruction Execution**. The model generates a response r_k for each instruction $instr_k$ without conditioning on any of the other instructions. We model this as:

$$p_\theta(r_k \mid p_r, x, instr_k), \quad \text{for each } k \in \{1, \ldots, n\}.$$

The final type is **Think and Answer Generation**. In the final step of inference the model conditions on all thread responses, does a final round of thinking c before generating the answer a. This is modeled as:

$$p_\theta(c, a \mid p_a, x, r).$$

, where r is all the thread responses concatenated together.

For each problem in our dataset, the training data consists of one instruction generation example (using the appropriate p_i), n parallel execution examples (each using p_r), and one result aggregation example (using p_a). To prevent overfitting on the more frequent parallel execution tasks, we scale the sampling probability for each execution data point by $1/n$. This ensures that for any given problem, there is an equal likelihood of sampling from each of the three task types.

4.4 Inference

Having discussed data generation and the functions that we model during training, we now discuss how these functions can be used to perform parallel reasoning at inference. Parallel inference starts with a diversity-first approach, optionally applying the decompose strategy for deeper, nested parallelism. The process begins with **Instruction Generation**. Given a problem x and the diversity prompt $p_{instr,1}$, the model first generates the number of threads, n, and a set of diverse, high-level instructions or strategies to solve the problem, $\{instr_1, \ldots, instr_n\}$, by drawing from their joint distribution, $p_\theta(n, i \mid p_{instr,1}, x)$. The number of threads, n, can also be set manually (we refer to this as thread-forcing).

With these high-level instructions established, we proceed to **Thread Instruction Execution**. For each instruction $instr_k$, we generate a corresponding response r_k, which can be accomplished in two ways. The first method is direct execution, where we generate the response immediately for each of the n instructions in parallel, drawing from the distribution:

$$r_k \leftarrow p_\theta(r_k \mid p_r, x, instr_k)$$

Alternatively, for potentially faster inference on complex instructions, we can apply the **Decomposition Operator**.

$$r_k = \text{DecomposeAndExecute}(x, instr_k)$$

This generates the response by recomposing the problem and instruction into parallelizable computations before it returns a response, see Appendix D for more details on the Decomposition Operator. Regardless of the method chosen, all n response-generation processes are performed concurrently . Finally, in the **Result Aggregation** stage, the model gathers the complete set of n responses, $\{r_1, \ldots, r_n\}$, It then produces the final summary and answer by drawing from the distribution:

$$(c, a) \leftarrow p_\theta(c, a \mid p_a, x, r).$$

5. Experiments

In our experiments, we aim to evaluate the model trained for parallel inference based on its ability to: 1. Decompose embarrassingly parallel problems into parallelized subtasks. 2. Demonstrate that the parallel dimension of reasoning over diverse approaches serves as a powerful test-time inference scaling axis. 3. Investigate how this model compares to other test-time scaling methods, evaluated both on a throughput basis, leveraging the assumption of freely available parallelizable computation.

5.1 Setup

We evaluate parallel reasoning capabilities through a series of experiments. First, we assess the model's ability to decompose problems using a held-out set of embarrassingly parallel synthetic problems. Next, we measure performance across multiple benchmarks, including AIME 2024 and 2025 (Art of Problem Solving, 2025), MATH-500 (Hendrycks et al., 2021), GSM8K (Cobbe et al., 2021), Countdown 3-to-4 (Pan, 2024), and Enigmata Eval (Chen et al., 2025a), using Qwen-3-4B-Parallel to investigate the scaling properties of thread count. Finally, we compare Qwen-3-4B-Parallel to Budget Forcing and Chain-of-Thought (CoT) approaches, evaluating generation time on a single H100 GPU with a batch size of 1 against accuracy. To ensure a fair comparison, we fine-tune a separate model, Qwen-3-4B-Grok-Distill, on the same problems using standard CoT reasoning traces generated by Grok-3-Mini. This controls for the potential impact of fine-tuning itself on performance.

5.2 Dataset Generation

We employ three main dataset types to train our parallel reasoning strategies. The S1K dataset (Muennighoff et al., 2025) a curated collection of mathematical problems, the Enigmata dataset (Chen et al., 2025a) for its diverse, non-mathematical problems that often require search and creative reasoning. Finally, to explicitly teach decomposition, we generate synthetic simple arithmetic problems that are inherently parallelizable where the number of subtasks is predefined. We give full details of datasets used in Appendix A.

5.3 Decompose Reasoning

We investigate the ability for Qwen-3-4b-parallel to recognize computations that can be executed in parallel by comparing the number of threads predicted by the model versus the intended number of synthetic sub tasks of the problem. We perform this evaluation on a held out subset of tasks not seen during training. Figure 2 shows a clear strong positive correlation between these two variables. This demonstrates that the student model has effectively learned the skill of spotting parallelism opportunities which is the critical skill needed for leveraging the decompose operator to speed up inference in Qwen-3-4b-parallel.

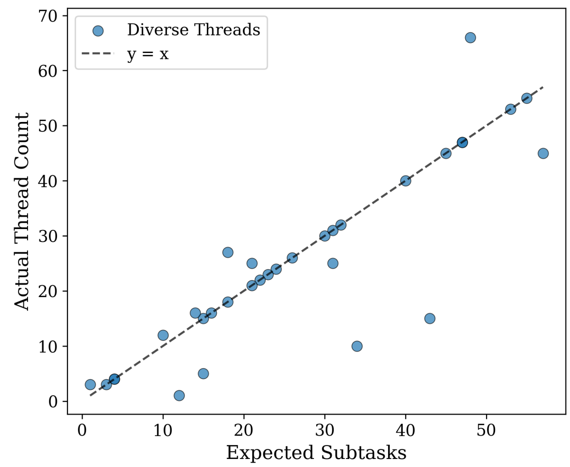

Figure 2: Decompose Operator evaluated on a dataset of synthetic calculation problems with known number of operations that can be parallelized over.

5.4 Timing

Since our parallelism approach is not fully parallelizable, due to still making use of sequential instruction generation and a thinking section as we scale the number of threads generation time increases. However this scaling is far less, when compared to budget forcing, which due to the quadratic complexity of transformers as sequence length increases generation time becomes very slow. In Figure 3 we compare how generation time scales as compute is scaled for budget forcing and thread forcing (forcing the model to generate fixed number of threads via variable n). Due to parallelism Qwen-3-4b-parallel has far better scaling properties when assessed on generation time, of course since it is also a transformer it still has quadratic complexity however since inference is split across n shorter sequences the cost of scaling to high token budgets is far less.

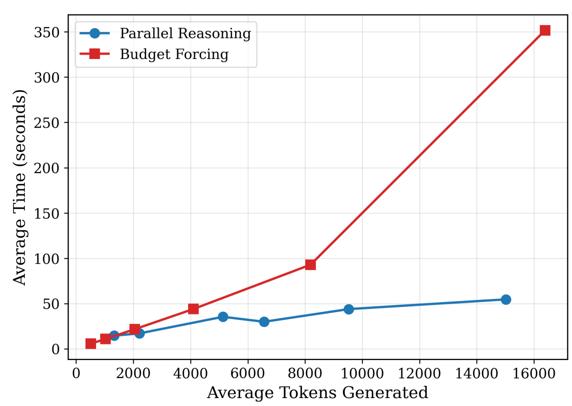

Figure 3: Average generation time for varying token budgets for Parallel Reasoning and Budget Forcing on Math 500 dataset

5.5 Strawberry Example

In order to give better intuitions on the computations performed by both the diversify and decompose reasoning strategies learned by Qwen-3-4b-parallel, we provide examples of generations, see Appendix H for additional generations. Firstly, we give an example of only the diversify operator being used. Table 1 shows the diversify operator being executed on a task that LLMs often struggle with. In this execution, we find that indeed some threads make errors in their approach to solve the problem. What is particularly interesting about this generation is that there is an even split across answers of 1, 2, and 3. Aggregation strategies such as majority voting would not lead to the correct answer in this case. However, since Qwen-3-4b-parallel has learned to aggregate the reasoning and analyze it, it is able to assess the different reasoning traces and decide what is most convincing, which in this case ultimately leads to generating the correct answer. We provide two additional generations in Appendix H where we also demonstrate the execution of the decompose operator on a mathematical calculation, showing its ability to identify parallelizable computations.

Problem: How many r's are there in the word strawberry?
Predicted number of threads: 6

N	**Instruction**	**Response**
1	Examine the word "strawberry" letter by letter and count each occurrence of the letter 'r' sequentially.	Start with the word "strawberry". Begin at the first letter: 's' is not 'r'. Next, 't' is not 'r'. Then, 'r' is 'r', so that's one. Continue: 'a' is not 'r'. 'w' is not 'r'. 'b' is not 'r'. 'e' is not 'r'. 'l' is not 'r'. 'l' is not 'r'. 'y' is not 'r'. The word has only one 'r' at the third position. No further 'r's are present. Thus, the count is one.
2	Spell the word "strawberry" and use positional analysis to identify and count the letter 'r' based on its spelling.	The word "strawberry" is spelled S-T-R-A-W-B-E-R-R-Y. By writing it out, I can visually [...] Position 1: S, not R. [...] I've identified three instances of 'R'. [...]
3	Break down "strawberry" into its root words or etymological components and check for the letter 'r' in those parts.	"Strawberry" originates from Old English "straewyrbe," combining "straew" (straw) and "yrbe" (apple). [...] Thus, two 'r's are evident. [...]
4	Write out "strawberry" and use a phonetic approach to mentally identify letters that correspond to the sound 'r'.	Phonetically, "strawberry" is pronounced as "struh-BRAH-fee". [...] This phonetic method confirms it's just one. [...]
5	Search for the letter 'r' in "strawberry" by eliminating letters one at a time and focusing on remaining ones.	Start with "s-t-r-a-w-b-e-r-r-y". Eliminate 's' first [...] Eliminate 'e', so "r-r-r-y". Now, we have three 'r's at the start. [...]
6	Think about common letters in the word "strawberry" and determine the presence and count of 'r' based on frequency patterns.	"Strawberry" is a common word with a mix of vowels and consonants. [...] Thus, there are two 'r's.

Aggregated thinking: <think>The threads provide diverse approaches to counting the 'r's in "strawberry." Thread 1 sequentially checks each letter, identifying one 'r.' Thread 2 spells out the word and counts systematically, finding three 'r's. Thread 3 uses etymology to split the word, revealing two 'r's. Thread 4 focuses on pronunciation, confirming one 'r.' Thread 5 eliminates letters to isolate 'r's, resulting in three 'r's. Thread 6 leverages common letters in English words, suggesting two 'r's. Insights show discrepancies: some threads count one, others three, or two. The most consistent evidence comes from spelling-based methods (Threads 2 and 5) and etymology (Thread 3), as they directly examine the word's structure. Pronunciation (Thread 4) and simplified approaches (Thread 1) may overlook some 'r's. **Aggregating, the accurate count is three 'r's, as verified by direct examination.**</think>

Final answer: <answer>**3** </answer>

Table 1: Diversify parallel reasoning execution example for the problem "How many r's are in strawberry?"

5.6 Performance

We find in Figure 4 that the thread dimension of our parallelism approach delivers strong scaling performance across a range of math and problem solving benchmarks. These results show that the Qwen3-4b-parallel out-performs Qwen-3-4b-Grok-distill both in terms of final performance but also in terms of faster inference speed, which enables steeper scaling with compute. Parallel inference also gains on the original base model. We also find that standard chain of thought inference is prone to generating very short chains resulting in very low performance but fast inference. In addition see Figure 5 for full scaling evaluation of Qwen3-4b-parallel across 6 different tasks showing stronger scaling performance in the thread dimension across all tasks .

6. Conclusion

In this work, we find that generating pseudo parallel data sequentially with a teacher model, and distilling it into a student model is a valid paradigm for teaching parallel reasoning. We demonstrated the effectiveness of targeting two operations: diversify and decompose. Diversify demonstrated strong scaling properties in the number of threads dimension, with performance consistently improving as the

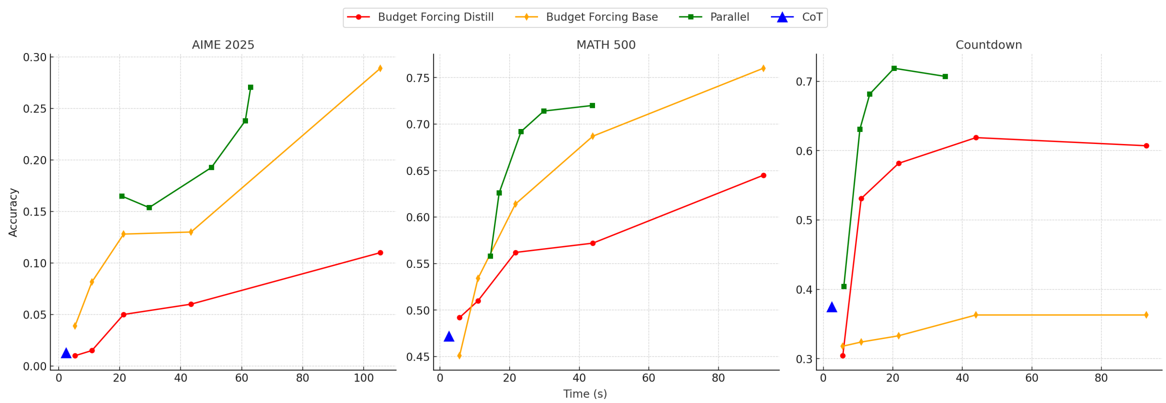

Figure 4: Budget forcing (max 8192 tokens) base and distill, Chain of Thought, Qwen-3-4b-parallel (max 16 threads)

number of parallel reasoning threads increased across various benchmarks, serving as a powerful test-time inference scaling axis. We also show that decompose reasoning is able to effectively predict the number of subtasks that problems can be parallelized over. When leveraged together our parallel reasoning approach exhibits faster inference compared to methods like budget forcing at test time while maintaining strong performance.

Limitations and Future Work

Our current approach has several limitations that suggest avenues for future work. Firstly we found low performance in the Grok fine-tune of Qwen3-4b which could be further investigated to ensure fair comparisons. Secondly, we did not investigate the model's ability to perform adaptive computation, as this would likely require some form of on-policy learning to correct for the differences between what the teacher and student model find difficult. Future work could also investigate extending our method with reinforcement learning to generate policy improvements beyond distillation alone.

7. Disclosure of Funding

Matthew Macfarlane is supported by the LIFT-project 019.011, which is partly financed by the Dutch Research Council (NWO). We also thank reviewers for their detailed feedback which we have integrated into the final version of the paper.

References

AbdElhameed, M. and Halim, P. Inference scaling vs reasoning: An empirical analysis of compute-optimal llm problem-solving, 2024.

Art of Problem Solving. 2025 AIME I, 2025. URL https://artofproblemsolving.com/wiki/index.php/2025_AIME_I.

Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., and Hoefler, T. Graph of thoughts: Solving elaborate problems with large language models, 2023.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling, 2024.

Chen, J., He, Q., Yuan, S., Chen, A., Cai, Z., Dai, W., Yu, H., Yu, Q., Li, X., Chen, J., et al. Enigmata: Scaling logical reasoning in large language models with synthetic verifiable puzzles. *arXiv preprint arXiv:2505.19914*, 2025a.

Chen, M., Hui, B., Cui, Z., Yang, J., Liu, D., Sun, J., Lin, J., and Liu, Z. Parallel scaling law for language models. *arXiv preprint arXiv:2505.10475*, 2025b.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems, 2021. *URL https://arxiv. org/abs/2110.14168*, 9, 2021.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, Ł. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Ding, Y., Jiang, W., Liu, S., Jing, Y., Guo, J., Wang, Y., Zhang, J., Wang, Z., Liu, Z., Du, B., Liu, X., and Tao, D. Dynamic parallel tree search for efficient llm reasoning, 2025.

Gandhi, K., Lee, D., Grand, G., Liu, M., Cheng, W., Sharma, A., and Goodman, N. D. Stream of search (sos): Learning to search in language. *arXiv preprint arXiv:2404.03683*, 2024.

Hao, S., Sukhbaatar, S., Su, D., Li, X., Hu, Z., Weston, J., and Tian, Y. Training large language models to reason in a continuous latent space, 2024.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Jin, T., Cheng, E. Y., Ankner, Z., Saunshi, N., Elias, B. M., Yazdanbakhsh, A., Ragan-Kelley, J., Subramanian, S., and Carbin, M. Learning to keep a promise: Scaling language model decoding parallelism with learned asynchronous decoding, 2025.

Kang, D., Zhang, Y., Song, D., Gonzalez, J. E., Klein, D., Goldberg, Y., Recht, B., and Hashimoto, T. B. Scalable best-of-n selection for large language models via self-certainty. *arXiv preprint arXiv:2403.02475*, 2024. URL https://arxiv.org/abs/2403.02475.

Liao, B., Dong, H., Xu, Y., Sahoo, D., Monz, C., Li, J., and Xiong, C. Fractured chain-of-thought reasoning. *arXiv preprint arXiv:2505.12992*, 2025.

Liu, J. S. and Chen, R. Sequential monte carlo methods for dynamic systems. *Journal of the American statistical association*, 93(443):1032–1044, 1998.

Muennighoff, N., Yang, Z., Shi, W., Li, X. L., Fei-Fei, L., Hajishirzi, H., Zettlemoyer, L., Liang, P., Candès, E., and Hashimoto, T. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

Ning, X., Lin, Z., Zhou, Z., Wang, Z., Yang, H., and Wang, Y. Skeleton-of-thought: Prompting llms for efficient parallel generation. *arXiv preprint arXiv:2307.15337*, 2023.

Pan, J. Countdown-tasks-3to4 dataset, 2024.

Pan, J., Li, X., Lian, L., Snell, C., Zhou, Y., Yala, A., Darrell, T., Keutzer, K., and Suhr, A. Learning adaptive parallel reasoning with language models. *arXiv preprint arXiv:2504.15466*, 2025.

Rodionov, G., Garipov, R., Shutova, A., Yakushev, G., Egiazarian, V., Sinitsin, A., Kuznedelev, D., and Alistarh, D. Hogwild! inference: Parallel llm generation via concurrent attention. *arXiv preprint arXiv:2504.06261*, 2025.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y., Wu, Y., et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models, 2022.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Wu, Y., Sun, Z., Li, S., Welleck, S., and Yang, Y. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models, 2024.

Wu, Y., Wang, Y., Du, T., Jegelka, S., and Wang, Y. When more is less: Understanding chain-of-thought length in llms. *arXiv preprint arXiv:2502.07266*, 2025.

xAI. Introducing grok 3: A new era of ai innovation, n.d. URL https://x.ai/news/grok-3. Accessed: 2025-06-21.

Yang, X., An, Y., Liu, H., Chen, T., and Chen, B. Multiverse: Your language models secretly decide how to parallelize and merge generation. *arXiv preprint arXiv:2506.09991*, 2025.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023.

Zhang, L., Hosseini, A., Bansal, H., Kazemi, M., Kumar, A., and Agarwal, R. Generative verifiers: Reward modeling as next-token prediction, 2024.

Zhao, S., Brekelmans, R., Makhzani, A., and Grosse, R. Probabilistic inference in language models via twisted sequential monte carlo. *arXiv preprint arXiv:2404.17546*, 2024.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36: 46595–46623, 2023.

A. Datasets

A.1 S1K

The **S1K** dataset is a collection of 1,000 questions, each accompanied by a detailed reasoning trace. It was specifically designed to enable strong reasoning performance with minimal training compute. The dataset was created by distilling questions and reasoning traces from the Google Gemini Flash Thinking API. The curation process started with an initial pool of 59,029 questions from 16 diverse and high-quality sources, including NuminaMATH, AIME (1983-2021), OlympicArena, OmniMath, and AGIEval. This larger set was then filtered down to 1,000 high-quality, difficult, and diverse samples. The filtering process involved removing API errors and low-quality examples, excluding questions that were easily solved by baseline models (Qwen2.5-7B-Instruct and Qwen2.5-32B-Instruct), and selecting for domain diversity across 50 categories based on the MSC system. Questions with longer reasoning traces were favored as an indicator of higher difficulty. The S1K dataset is open-source and has demonstrated significant sample efficiency for training reasoning models. In this work we do not leverage the reasoning traces from the S1K dataset but use the curated problem set that represents a wide covering but also curated set of problems that can be used to teach LLMs to reason.

Table 2: S1K Dataset Overview

Factor	Description
Size	1,000 questions with reasoning traces and answers.
Curation Principles	Quality, Difficulty, and Diversity.
Initial Pool Size	59,029 questions.
Number of Domains	50 unique domains (e.g., Geometry, Number Theory, Biology).

A.2 Enigmata

The **ENIGMATA** suite is a comprehensive framework designed to enhance the logical reasoning abilities of Large Language Models (LLMs), particularly in solving puzzles. Unlike traditional reasoning benchmarks that often lack diversity or scalability, ENIGMATA features 36 distinct puzzle tasks across seven broad categories: Crypto, Arithmetic, Logic, Grid, Graph, Search, and Sequential puzzles. Each task is equipped with a unique auto-generator capable of producing unlimited examples with controllable difficulty and a rule-based verifier for automatic evaluation. For this benchmark the generators to generate training data for each problem has not been made public. Therefore in the current version of the paper we split the ENIGMATA dataset into two sub-datasets sampled randomly across tasks with the same distribution of tasks for each dataset. We generate a train and evaluation dataset both of size 2048.

Table 3: ENIGMATA Suite Overview

Factor	Description
Tasks	36 unique puzzle tasks.
Categories	7 (Crypto, Arithmetic, Logic, Grid, Graph, Search, Sequential).
Verification	Rule-based auto-verifiers for all 36 tasks.
Original Benchmark	4,758 puzzle instances.
Training Dataset (this paper)	2048 puzzle instances.
Evaluation Dataset (this paper)	2048 puzzle instances.

A.3 Synthetic Parallel

The **Synthetic Parallel** dataset was created to produce complex problems that are inherently decomposable into multiple, independent sub-problems. This design facilitates the training and evaluation of a large language model's ability to perform parallel reasoning. The dataset was generated using the `x_grok-3-mini-beta` model.

The generation process involves two main stages. First, a problem is synthetically created. The model is prompted to design a complex problem from one of **nine distinct problem types**: calculation, algorithm, optimization, simulation, sorting, matrix, calculator, one-step simulation, and n-body problems. The complexity and the number of inherent sub-tasks (ranging from 1 to 64) are specified during this creation phase. The prompt explicitly instructs the model to avoid any language that would overtly suggest a parallel approach to solving the problem, ensuring that the decomposable nature of the problem is a natural characteristic of the problem itself.

In the second stage, the same `x_grok-3-mini-beta` model is used to solve the generated problem using a structured, parallel reasoning approach. The model is prompted to first break the problem down into a number of distinct reasoning "threads." For each thread, it must generate a concise and unique instruction. Following the instructions, the model provides the reasoning for each thread independently, enclosed in specific XML tags (e.g., <thread_n_instruction> and <thread_n>). This structured output allows for the clear separation and analysis of each reasoning path. A key feature of our data generation is the ability to force the model to use a specific number of threads, which allows for the study of how the model performs under such constraints. The generated solutions are then automatically evaluated to ensure they adhere to the required format. This process yielded a dataset of 128 samples.

Table 4: Synthetic Parallel Dataset Overview

Factor	**Description**
Size	128 problems with structured parallel solutions.
Generation Model	`x_grok-3-mini-beta` for both problem and solution generation.
Problem Types	9 (e.g., Calculation, Optimization, Simulation).
Sub-tasks	A range of 1 to 64 independent sub-tasks per problem.
Reasoning Structure	Explicit instructions and reasoning for each parallel thread.

B. Data Generation

This section details the data generation methodology, specifying the configurations for the primary generation model and the subsequent evaluation models. The process involves a main generation step followed by distinct validation stages for correctness and independence, each utilizing a specifically tuned model. The entire pipeline leverages API calls to X.AI models.

B.1 Main Generation ("Diverse Approaches")

The core synthetic dataset was produced using the **x_grok-3-mini-beta** model. The parameters were configured to accommodate extensive parallel reasoning tasks.

Table 5: Hyperparameters for Main Data Generation

Parameter	Value
Model	x_grok-3-mini-beta
Max Tokens	100,000
Temperature	0.3 (default)
Top-p	1.0 (default)

B.1.1 Prompts

System Prompt template

```
You are a helpful assistant solving problems using a parallel reasoning process to explore
    diverse ways to solve a problem, implemented sequentially. Follow this exact structure:
1. Generate {num_threads} distinct thread instructions, each concise (<=100 words), unique
    in approach that has a chance of solving the problem. Output each as <
    thread_n_instruction> </thread_n_instruction>.
2. For each thread, provide reasoning based only on the problem and its specific
    instruction, using around 40 sentences to explore the assigned direction, enclosed in <
    thread_n> </thread_n> tags.
3. Aggregate insights from all threads in a <think> </think> section, synthesizing their
    reasoning.
4. Provide the final answer in an <answer> </answer> section.
5. The final answer should only be a concise answer, if the answer is a number give only
    the number and no filler text, if it is a string or proof give the string or proof and
    no additional filler text. Start with thread instructions, followed by thread reasoning
    , then aggregation, then the answer. Ensure clarity in instructions. The instructions
    must be diverse to spread the load of the reasoning and prevent redundancy.
    Instructions must be independent of each other and not rely on each other's reasoning
    or results. The threads should not condition on each other's results or reasoning in
    any way to ensure they can be evaluated independently. Do not make use of <result> </
    result> tags at all, you should only ever use code block tags, if they are mentioned in
     the problem, they are only used as placeholders for the answer in the question, they
    are not to be used at generation time. Use code block tags for the solution. Follow the
     exact format the problem asks for the answer to be given in, for example if matrices
    are specified in list of list format do that or using \n tags then use those.
```

User Prompt template

```
Problem: {problem}
Use code block tags for the solution. Do not use <result> tags at all, they are only used
    as placeholders for the answer in the question, they are not to be used at generation
    time. Follow the exact format the problem asks for the answer to be given in, for
    example if matrices are specified in list of list format do that or using \n tags then
    use those.
```

B.2 Validation and Independence Checks

To ensure the integrity and quality of the generated data, we implemented a two-fold validation procedure using the more capable **x_grok-3-beta** model. This process assessed both the correctness of the generated answers and the statistical independence of the instructions intended for parallel execution.

B.2.1 LLM Answer Evaluation

For correctness validation, we configured the model to act as an evaluator. It was prompted to compare the generated answer against a ground truth, responding with a simple binary classification ("1" for correct, "0" for incorrect). A higher temperature was used to provide more nuanced judgment.

Table 6: Hyperparameters for LLM Answer Evaluation

Parameter	**Value**
Model	`x_grok-3-beta`
Max Tokens	10
Temperature	0.3
Top-p	1.0

System Prompt template

```
You are an evaluator. Given a problem, a ground truth answer, and a generated answer,
    determine if the generated answer is correct. The generated answer does not need to be
    identical to the ground truth answer. For example, in problems like countdown, there
    can be multiple ways to solve the problem. Respond with only '1' if correct and '0' if
    incorrect. Ensure your response contains *only* the digit '1' or '0' and nothing else.
```

User Prompt template

```
Problem: {problem} Ground Truth Answer: {ground_truth_answer} Generated Answer: {
    generated_answer} Is the Generated Answer correct? Respond with only '1' for correct or
     '0' for incorrect.
```

B.2.2 Parallel Instruction Independence Evaluation

To verify that problem instructions could be executed independently in parallel, a separate evaluation step was performed. The model was prompted to assess a set of instructions and determine if they were free from dependencies, again providing a binary response.

Table 7: Hyperparameters for Instruction Independence Evaluation

Parameter	Value
Model	`x_grok-3-beta`
Max Tokens	10
Temperature	0.3
Top-p	1.0

System Prompt template

```
You are an evaluator assessing whether computational thread instructions can be executed
    independently and in parallel. For instructions to be independent, each thread must be
    able to complete its work without needing results from other threads. Threads should
    not have sequential dependencies. Respond with ONLY '1' if all instructions are
    independent and can be executed in parallel, or '0' if any instructions depend on
    results from other threads. Your response must contain ONLY the digit '1' or '0' and
    nothing else.
```

User Prompt template

```
Evaluate whether the following thread instructions can be executed independently and in
    parallel. The instructions should not depend on each other's results or computations.

Thread Instructions:
{thread_instructions}

Are all these instructions independent and can be executed in parallel without waiting for
    each other's results? Respond with ONLY '1' for yes (completely independent) or '0' for
     no (some dependencies exist).
```

C. Data Generation Statistics

This section provides a statistical overview of the data generated using both parallel and Chain of Thought (CoT) methodologies. The correctness of generated solutions was evaluated for each approach across different datasets.

C.1 Parallel Generation

This subsection presents the statistics for data generated using the parallel reasoning approach, evaluating performance across various thread configurations. Each table details the performance for a specific dataset, with overall metrics and correctness percentages for each thread count organized as individual rows.

C.1.1 ENIGMATA DATASET

Table 8: Statistics for Parallel Generation on Enigmata

Metric	Value
Total Problems	2500
Total Correct Problems	553
1 Thread	74.14%
2 Threads	12.12%
4 Threads	4.70%
8 Threads	4.88%
16 Threads	1.99%
32 Threads	1.99%
64 Threads	0.18%

C.1.2 S1K DATASET

Table 9: Statistics for Parallel Generation on S1K

Metric	Value
Total Problems	1000
Total Correct Problems	653
1 Thread	48.10%
2 Threads	6.20%
4 Threads	3.60%
8 Threads	2.50%
16 Threads	2.40%
32 Threads	1.50%
64 Threads	1.00%

C.1.3 SYNTHETIC DATASET

For the synthetic dataset problems that are decomposable were generated by ”grok-3-mini” using the prompt. The number of sub-tasks were sampled uniformly in the range (1-64).

C.2 Chain of Thought Generation

This subsection details the statistics for data generated using the traditional Chain of Thought approach.

C.2.1 ENIGMATA DATASET

Table 10: Statistics for Chain of Thought Generation on Enigmata

Metric	Value
Total Problems	2048
Total Correct Problems	400

C.2.2 S1K DATASET

Table 11: Statistics for Chain of Thought Generation on S1K

Metric	**Value**
Total Problems	1000
Total Correct Problems	579

D. Parallel Inference

D.1 Decompose and Execute

The ‘DecomposeAndExecute‘ operator provides a mechanism for nested parallelism, breaking down a single, complex reasoning instruction into smaller, parallelizable sub-problems. This process mirrors the main parallel inference framework but operates on a sub-task level.

Given a problem x and a high-level instruction $instr_k$ from the initial diversity step, the ‘DecomposeAndExecute‘(x, $instr_k$) function executes the following three stages:

1. **Sub-Instruction Generation**: The operator first recomposes the problem and the high-level instruction into a new prompt for decomposition. Using this, it generates a set of m sub-instructions, $\{j_1, \ldots, j_m\}$, by drawing from the distribution:

$$(m, j) \leftarrow p_\theta(m, j \mid p_{i,2}, x, instr_k)$$

Here, $p_{i,2}$ is the specific prompt for decomposition instruction generation. This step effectively breaks down the single complex task $instr_k$ into multiple, simpler, and concurrently executable sub-tasks.

2. **Parallel Sub-Task Execution**: With the sub-instructions established, the operator executes them in parallel to generate a set of sub-responses, $\{o_1, \ldots, o_m\}$. Each sub-response o_l is generated by conditioning on the problem, the original high-level instruction, and the corresponding sub-instruction j_l:

$$o_l \leftarrow p_\theta(o_l \mid p_r, x, instr_k, j_l), \quad \text{for each } l \in \{1, \ldots, m\}$$

Note that we use the same parallel response prompt, p_r, as in the main execution stage.

3. **Sub-Result Aggregation**: Finally, the operator aggregates the set of sub-responses, o, to produce the final, consolidated response r_k for the original high-level instruction $instr_k$. This is achieved by drawing from the aggregation distribution, now conditioned on the sub-responses:

$$r_k \leftarrow p_\theta(c, a \mid p_a, x, instr_k, o)$$

This aggregated result, r_k, is then passed back to the main diversity-level **Result Aggregation** stage, which synthesizes it along with all other parallel thread responses to produce the final answer.

E. Baselines

E.1 Chain of Thought (CoT)

Chain of Thought (CoT) is an inference technique where a model generates a sequence of intermediate reasoning steps before providing a final answer. For a given problem x, the model produces a reasoning trace (analogous to a single response r) which culminates in an answer a. This process can be denoted as $(r, a) = M_\theta(x)$. This approach improves performance on complex tasks by breaking them down into smaller, more manageable steps.

E.2 Budget Forcing

Budget Forcing is a strategy to control the computational resources allocated to a model's response. This is achieved by setting an explicit budget, B, which constrains a parameter such as the maximum number of tokens for the output. For a given problem x and a specified budget B, the model M_θ is prompted to generate a reasoning trace r and a final answer a, conditioned on this constraint. The generation process is denoted as $(r, a) = M_\theta(x \mid B)$, where the total length of the generated response is limited by the budget. This technique is used to study the trade-off between performance and computational cost and to analyze model behavior under explicit resource limitations.

E.3 Majority Voting

Majority Voting, also known as self-consistency, is an ensembling method used to determine a final answer from multiple reasoning paths. Given n paths generated (e.g., via thread forcing), which yield a set of answers $\{a_1, a_2, \ldots, a_n\}$, the final answer, a_{final}, is selected by choosing the most frequent one:

$$a_{\text{final}} = \underset{a'}{\operatorname{argmax}} \sum_{i=1}^{n} \mathbb{I}(a_i = a')$$

where $\mathbb{I}$ is the indicator function. This technique improves accuracy by filtering out sporadic errors and amplifying the correct answer, which is assumed to appear more consistently across different valid reasoning approaches.

F. Hyperparameters

This section details the hyperparameters used for the training and inference of our models.

F.1 Training

F.1.1 CHAIN OF THOUGHT

Table 12: Training Hyperparameters for Chain of Thought

Hyperparameter	**Value**
Model Name/Size	Qwen 3-4b
Batch Size	16
Learning Rate	1×10^{-5}
Learning Rate Scheduler	Constant with Warmup
Warmup Steps	100
Weight Decay	0.1
Adam Beta1	0.9
Adam Beta2	0.95
Adam Epsilon	1×10^{-8}
Gradient Clipping	1.0
Number of Epochs	1
Max Input Sequence Length	40960

F.1.2 PARALLEL REASONING

Table 13: Training Hyperparameters for Parallel Reasoning

Hyperparameter	**Value**
Model Name/Size	Qwen 3-4b
Batch Size	16
Learning Rate	1×10^{-5}
Learning Rate Scheduler	Constant with Warmup
Warmup Steps	100
Weight Decay	0.1
Adam Beta1	0.9
Adam Beta2	0.95
Adam Epsilon	1×10^{-8}
Gradient Clipping	1.0
Number of Epochs	1
Max Input Sequence Length	40960

F.2 Inference

F.2.1 Chain of Thought

Table 14: Inference Hyperparameters for Chain of Thought

Hyperparameter	Value
Temperature	0.6
Top-p (Nucleus Sampling)	0.95
Max Tokens	100k
Number of Generated Sequences	1

F.2.2 Budget Forcing

Table 15: Inference Hyperparameters for Budget Forcing

Hyperparameter	Value
Number of Generated Responses	1
Max Tokens per Response	[512, 1024, 2048, 4096, 8192]
Temperature	0.6
Top-p (Nucleus Sampling)	0.95

F.2.3 Majority Voting

Table 16: Inference Hyperparameters for Majority Voting

Hyperparameter	Value
Number of Generated Samples (k)	[1, 3, 5, 7 ,9, 11, 13]
Temperature	0.6
Top-p	0.95
Max Tokens	1024

F.2.4 Parallel Reasoning

Table 17: Inference Hyperparameters for Parallel Reasoning

Hyperparameter	Value
Number of Parallel Reasoning Paths	[1,2,4,8,16]
Temperature	0.6
Top-p (Nucleus Sampling)	0.95
Max Tokens per generation	8000

G. Further Experiments

G.1 Scaling

In this section, we investigate the scaling behavior of the Diversify Operator across a range of benchmarks, focusing on the impact of increasing the number of threads as a key scaling axis. Figure 5 illustrates the performance of the Diversify Operator as the compute is scaled along the thread dimension across six diverse evaluation datasets. The results demonstrate a consistent trend of improved performance with increased thread counts, highlighting the operator's ability to effectively leverage parallelization.

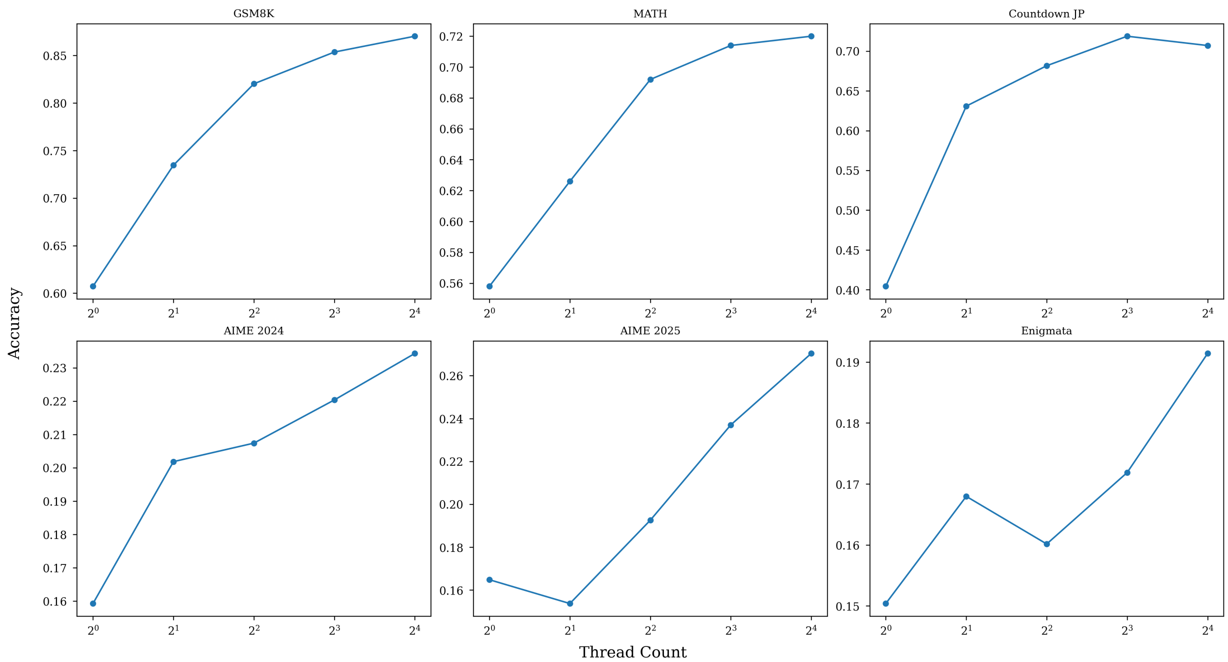

Figure 5: Diversify Operator reasoning compute scaled on the number of threads dimension for 6 evaluation datasets.

H. Parallel Reasoning Examples

In addition to our experimental results, we also provide examples of both the diversity operator and the decomposition operator. For examples of the diversity operator, we chose Fermi problems as a good choice. We chose these since most Fermi problems have no ground truth answer and, therefore, are amenable to different approaches to solve the problem. For decomposition tasks, we use examples of synthetic tasks that have inherently parallelizable operations, such as calculations that can be parallelized.

Problem: Estimate the total number of pizzas ever eaten.
Predicted number of threads: 4

N	Instruction	Response
1	Start with pizza production statistics: Calculate the total number of pizzas ever made by multiplying average daily production worldwide by the number of days since pizza's origin, assuming continuous production.	Pizza production statistics: Assume an average of 1 million pizzas made daily globally. Since pizza's origin in the 18th century (about 250 years), that's roughly 92,500 days. Multiplying 1 million by 92,500 gives about 92.5 billion pizzas. This is a simplified estimate based on production rates. Historical data from sources like the Pizza Chain shows growth, so adjusting for increases might add 20-30 billion more. Thus, the total could range from 112.5 to 122.5 billion. This method focuses on output figures. It's a production-centric view.
2	Begin with dietary data: Estimate the total number of pizzas ever eaten by analyzing global dietary consumption statistics, focusing on pizza as a regular meal for specific demographics.	Dietary data indicates that pizza is a staple in certain regions, like Italy and the U.S., where it's consumed regularly. Global dietary surveys suggest that millions of people eat pizza weekly, with estimates of 100 million people consuming it at least once a month. If we assume an average of 2 pizzas per person per year for this group, it totals around 200 million pizzas annually. Extending this to the entire history, say 10,000 years, gives 2 billion pizzas. This approach uses dietary patterns to extrapolate consumption, focusing on frequency and demographics. It doesn't account for non-dietary factors like cultural events.
3	Use cultural event data: Estimate pizza consumption based on data from major events like the World Pizza Championships, extrapolating to total historical eats through participation and attendance records.	Cultural events like the World Pizza Championships, with over 500 participants globally since 1999, could be extrapolated to total eats by assuming similar events occurred in local variations. If each event consumes 100 pizzas and there are 10,000 such events, that's 1,000,000. This is a small fraction of total. Attendance at these events might be in the thousands annually, but it's not comprehensive. Overall, this approach yields a conservative estimate of around 500,000 to 1,000,000.
4	Inspect waste and leftovers: Estimate the total number of pizzas ever eaten by analyzing data on pizza waste, leftovers, and disposal to infer consumption rates globally.	Pizza waste and leftovers provide a tangible measure of consumption. Globally, millions of tons of food waste are generated annually, with pizza being a common item. [....] If we estimate that 10% of all pizzas produced end up as waste or leftovers, and considering global production, this could add up to billions of pizzas discarded. For example, if 1 billion pizzas are wasted each year worldwide, and assuming a 10% waste rate from total production, that implies around 10 billion pizzas produced annually, leading to a cumulative total over history. This method uses environmental data to extrapolate consumption, treating waste as a proxy for total intake. It's a bottom-up approach that complements direct consumption data.

Aggregated thinking: <think>Thread 1 (production) provides the highest estimate, ranging from 112.5 to 122.5 billion pizzas. Thread 2 (dietary data) gives a much lower estimate of 2 billion pizzas, but the assumption of 10,000 years for pizza history is significantly longer than commonly accepted (18th century, 250 years) [...]. The waste data (Thread 4) also points to a very large number, on the order of tens of billions. Therefore, a robust estimate should fall within this higher range.</think>

<answer>**Around 100 billion pizzas** </answer>

Table 18: Decompose parallel reasoning execution example for pizza consumption problem. Use of [...] compression to remove long verbose text.

Problem: Calculate this mathematical expression:
$(((5 \times 5 \times 5 \times 5) + (99 \times 98))/((10000/25) - (\sqrt{1000000}))) \times (((2^{10}) - (50!/48!)) + (123 + 456 + 789))$
Predicted number of threads: 6

N	Instruction	Response
1	Calculate the value of (5×5×5×5) and determine how it contributes to the overall expression.	5×5×5×5 is calculated as 5 cubed times 5, which is 125 times 5, resulting in 625. This value is added to 99×98 in the numerator, so it contributes to the total sum before division. In the context of the entire expression, 625 helps form the numerator's base. This step is straightforward multiplication and powers. It sets the stage for the subsequent addition. The result is a key component in evaluating the overall fraction.
2	Compute the result of (99×98) and assess its role in the first part of the expression.	The expression (99×98) involves multiplying two consecutive integers. First, calculate 99×98: 99×98=9702. This value is added to 5×5×5×5 in the numerator. The result, 9702, contributes to the overall magnitude of the first part of the expression. In the full formula, this addition scales the numerator significantly. Thus, 9702 plays a key role in determining the final outcome.
3	Evaluate the denominator (10000/25) and subtract ($\sqrt{1000000}$) to find its value.	The denominator is (10000/25) minus $\sqrt{1000000}$. First, divide 10000 by 25 to get 400. Then, $\sqrt{1000000}$ is 1000. Subtract 1000 from 400 to get -600. This negative result is crucial for the overall fraction. The operation highlights the subtraction's impact. It sets the base for the division in the main problem. This step ensures accuracy in the calculation.
4	Calculate (2^{10}) and explore its impact on the second part of the expression.	(2^{10}) is 1024, a standard exponentiation result. This value is used in the subtraction with 50!/48!. It contributes to the overall magnitude of the second part. In mathematical contexts, powers like 2^{10} are common in computing and algebra. This calculation is straightforward and independent. It sets a base for further operations in the expression.
5	Compute (50!/48!) and analyze how it fits into the second part of the expression.	50!/48! equals 50×49×48!/48!=50×49=2450. This simplifies the factorial division, showing how factorials can be reduced. In the context of the full expression, it's a key subtraction term. This computation is straightforward once the factorial property is applied. It helps in evaluating the overall arithmetic.
6	Sum the values 123, 456, and 789, and determine their combined contribution.	Summing 123, 456, and 789: 123+456=579, then 579+789=1368. This is a straightforward addition of three numbers. The result is 1368, which is added to the other components in the expression. This step provides a base for the final calculation.

Aggregated thinking: <think>Thread 1 provides 5×5×5×5=625 for the numerator. Thread 2 adds 99×98=9702 to this, so numerator is 625+9702=10327. Thread 3 calculates the denominator as -600. Division gives $10327/-600 \approx -17.2117$. Thread 4 gives 2^{10}=1024, and Thread 5 gives 50!/48!=2450, so $1024-2450=-1426$. Thread 6 provides 123+456+789=1368, so add to -1426: $-1426+1368=-58$. Final multiplication: $-17.2117 \times -58 \approx 998.2886$, rounded to the nearest whole number as per context.</think>

<answer>**998** </answer>

Table 19: Decompose parallel reasoning execution example for calculation problem