

1 A Limitations

2 While Multiverse offers a general framework for generative modeling, its application to diverse data
 3 and task types beyond LLM reasoning remains underexplored. Moreover, as Multiverse-32B was
 4 trained solely using Supervised Fine-Tuning (SFT), a key challenge in future research is to integrate
 5 Reinforcement Learning (RL) into training, requiring the enhanced robustness of Multiverse engine.

6 B Broader Impacts

7 Multiverse enhances GPU utilization through parallel generation, which is particularly beneficial
 8 for small batch sizes and long-context inference. This approach significantly reduces latency and
 9 energy consumption. Furthermore, Multiverse enables economies of scale for many parallelizable
 10 tasks, decreasing the time per task unit while maintaining near-constant overall latency, even as task
 11 complexity increases. This scalability shows its potential to address extremely complex tasks that
 12 were previously intractable in practice, offering a promising path to artificial superintelligence (ASI).

13 C Theoretical Analysis

14 In this section, we provide an informal complexity analysis to showcase the scalability of Multiverse.

15 C.1 Problem Definition

16 Theoretically, we highlight the considerable scalability of Multiverse in addressing computationally
 17 intensive NP-complete tasks. By parallelizing subtasks and merging outcomes, it achieves notable
 18 time and space efficiency on such tasks, as exemplified by the **generalized, no-clue Sudoku** problem.

19 **Definition C.1** (Generalized, No-Clue Sudoku). This problem denote a tuple $\mathcal{S} = (G, V, \Sigma_0)$, where:

- 20 1. **Grid (G):** G is defined as an $N \times N$ matrix of cells, where $N = n^2$ for some integer $n \geq 1$. Each
 21 cell is indexed by a pair (i, j) with $1 \leq i, j \leq N$, corresponding to its row and column positions.
- 22 2. **Values (V):** $V = \{1, 2, \dots, N\}$ is the set of permissible values that can be assigned to each cell.
- 23 3. **Initial Configuration (Σ_0):** $\Sigma_0 : G \rightarrow \{\text{empty}\}$ is a function that assigns all cells in G an “empty”
 24 state. This signifies that there are no pre-assigned values since it is a “no-clue” Sudoku problem.
- 25 4. **Constraints:** A valid solution to this problem is a complete assignment $S : G \rightarrow V$ such that
 26 every cell $(i, j) \in G$ is assigned a value $S(i, j) \in V$, and the following constraints are satisfied:

- **Row Constraint (C_R):** For each row $i \in \{1, \dots, N\}$, the set of values $\{S(i, j) \mid 1 \leq j \leq N\}$ must be equal to V . That is, each value in V must appear exactly once in each row.

$$\forall i \in \{1, \dots, N\}, \forall v \in V, \exists! j \in \{1, \dots, N\} \text{ such that } S(i, j) = v$$

- **Column Constraint (C_C):** For each column $j \in \{1, \dots, N\}$, the set of values $\{S(i, j) \mid 1 \leq i \leq N\}$ must be equal to V . That is, each value in V must appear exactly once in each column.

$$\forall j \in \{1, \dots, N\}, \forall v \in V, \exists! i \in \{1, \dots, N\} \text{ such that } S(i, j) = v$$

- **Block Constraint (C_B):** The $N \times N$ grid G is partitioned into N disjoint $n \times n$ blocks. Let B_k denote the k -th block, for $k \in \{1, \dots, N\}$. For a block B_k , the set of values $\{S(i, j) \mid (i, j) \in B_k\}$ must be equal to V . That means each value in V must appear exactly once in each block.

$$\forall k \in \{1, \dots, N\}, \forall v \in V, \exists! (i, j) \in B_k \text{ such that } S(i, j) = v$$

- 27 5. **Objective:** The objective of this problem is to find *any* complete assignment $S : G \rightarrow V$ that
 28 satisfies all constraints. Since Σ_0 provides no clues, the task is to construct a valid Sudoku grid of
 29 size $N \times N$ from an entirely empty state. We only determine if at least one valid solution exists.

30 C.2 Sequential Generation

31 We begin by analyzing the time and space complexity of solving such tasks with sequential generation.

Theorem C.2 (Worst-case Time Complexity of Sequential Sudoku Solver). *Let $T(k)$ be the worst-case time required by a depth-first search (DFS) algorithm to decide (i.e., find a solution or prove unsatisfiability) a generalized Sudoku problem with k empty cells on an $n^2 \times n^2$ grid. Then, we have*

$$T(k) = \Theta\left((n^2)^k\right),$$

and in particular for $k = n^4$ (the non-clue one with an entirely empty grid),

$$T(n^4) = \Theta\left((n^2)^{n^4}\right).$$

Proof. We show both an O -upper bound and an Ω -lower bound.

Upper bound. We induct on k , the number of empty cells to be filled, to derive an upper bound.

- Base case $k = 0$: $T(0) = \Theta(1)$. If $k = 0$, all necessary cells have been assigned. The process of recognizing that a complete assignment has been made (i.e., a leaf in the search tree is reached) and proceeding accordingly (e.g., returning success) takes constant time at this stage of the recursion.
- Inductive step: Assume $T(k - 1) = O((n^2)^{k-1})$ for $k > 0$. When k cells are empty, the DFS algorithm selects one empty cell. It then tries at most n^2 possible values for this cell. For each value, its validity with respect to the current partial assignment is checked (assumed to be $O(1)$), and the algorithm then recursively calls itself for the remaining $k - 1$ empty cells. Thus, we have:

$$T(k) \leq n^2 \cdot T(k - 1) + O(1).$$

Substituting the inductive hypothesis:

$$T(k) \leq n^2 \cdot O((n^2)^{k-1}) + O(1) = O((n^2)^k) + O(1) = O((n^2)^k).$$

By induction, $T(k) = O((n^2)^k)$ for all $k \geq 0$.

Lower bound. We consider examples where the DFS algorithm cannot prune branches early.

- For $k = 0$, $T(0) \geq \Omega(1)$. This is consistent with the base case of the upper bound.
- Suppose for some $k \geq 1$, the instance is constructed such that any partial assignment of fewer than k cells appears extendable, and unsatisfiability is only determined once all k cells are assigned. Such “hard” instances can be created by introducing a global XOR-like constraint over all k cells that is satisfiable on exactly one completion but remains consistent with every proper partial assignment. Concretely, add a clause that is violated only when all k cells are assigned (e.g. the parity of their values). In such a scenario, the solver must try each of the n^2 choices for the first of the k cells (as none can be ruled out prematurely). For each such choice, it must perform at least $T(k - 1)$ work on the subproblem involving the remaining $k - 1$ cells. Hence, the recurrence is:

$$T(k) \geq n^2 \cdot T(k - 1) + \Omega(1).$$

By induction, this unfolds to:

$$T(k) \geq (n^2)^k T(0) + \Omega\left(\sum_{i=0}^{k-1} (n^2)^i\right) = \Omega((n^2)^k).$$

Tightness. Combining the upper and lower bounds yields $T(k) = \Theta((n^2)^k)$. □

Theorem C.3 (Worst-case Space Complexity of Sequential Sudoku Solver). *Let $S(k)$ be the worst-case space required by a DFS solver on a generalized Sudoku problem with k empty cells. To be consistent to sequential generation in LLMs, it should address this problem with the following rules:*

1. *explores children in a fixed order,*
2. ***stops** descending a branch as soon as the current partial assignment violates a constraint*
3. ***never frees** any of the partial assignments it has generated.*

65 Then, we have

$$S(k) = \Theta(k(n^2)^k).$$

66 and in particular for $k = n^4$ (the non-clue one with an entirely empty grid),

$$S(n^4) = \Theta(n^4(n^2)^{n^4}).$$

67 *Proof.* We show both an O -upper bound and an Ω -lower bound.

68 **Upper bound.** At depth i ($0 \leq i \leq k$) the algorithm can generate at most $(n^2)^i$ distinct partial
 69 assignments (one for each choice of values along the path from the root). Each such node stores
 70 exactly i cell-value pairs, so the total number of stored pairs is bounded by

$$\sum_{i=0}^k i(n^2)^i = \Theta(k(n^2)^k).$$

71 Hence $S(k) = O(k(n^2)^k)$.

72 **Lower bound.** Construct a family of no-clue instances that postpones *all* contradictions until level k .
 73 Fix an ordering of the k decision cells c_1, \dots, c_k and add a single global XOR clause

$$\bigoplus_{j=1}^k (X_{c_j,1} \vee X_{c_j,2}) = 1,$$

74 which is violated only after every c_j is assigned. Thus every branch reaches depth k before any conflict
 75 is detected, forcing the algorithm to materialise the entire (n^2) -ary tree of depth k . Consequently it
 76 must store exactly $\sum_{i=0}^k i(n^2)^i = \Theta(k(n^2)^k)$ cell-value records, giving $S(k) = \Omega(k(n^2)^k)$.

77 **Tightness.** The upper and lower bounds coincide, so $S(k) = \Theta(k(n^2)^k)$. □

78 C.3 Parallel Generation

79 Following the definition, we further examine the time and space complexity of solving such tasks
 80 with parallel generation. For a practical analysis, we denote P_{\max} as the maximum parallelism limit.
 81 We analyze the simplest strategy \mathcal{P} that launches one branch per *complete* assignment of the n^4 cells:

$$B_{\text{cell}} = (n^2)^{n^4} = n^{2n^4} \text{ independent branches.}$$

82 We can execute at most P_{\max} branches *in parallel*. Any further branches are queued into later batches.
 83 Based on these requirements, we analyze the time and space complexity of parallel generation.

Theorem C.4 (Worst-case Time Complexity of Parallel Sudoku Solver). *Let $T_{\mathcal{P}}(P_{\max})$ be the worst-case number of batches required by strategy \mathcal{P} under a concurrency cap $P_{\max} \geq 1$. Then*

$$T_{\mathcal{P}}(P_{\max}) = \left\lceil \frac{B_{\text{cell}}}{P_{\max}} \right\rceil \text{ with } B_{\text{cell}} = (n^2)^{n^4}.$$

84 *Proof.* We show both an O -upper bound and an Ω -lower bound.

85 **Upper bound.** Each batch can host at most P_{\max} branches. Hence the solver finishes after at most
 86 $\lceil B_{\text{cell}}/P_{\max} \rceil$ batches.

87 **Lower bound.** Construct an adversarial *unsatisfiable* instance by adding a single global parity (XOR)
 88 clause that is violated *only when every variable is fixed*. Therefore *every* complete assignment must
 89 be explored to discover the contradiction; no branch halts prematurely. Because each batch solves at
 90 most P_{\max} branches in parallel, at least $\lceil B_{\text{cell}}/P_{\max} \rceil$ batches are necessary.

91 **Tightness.** Upper and lower bounds coincide, so the equality holds. □

Theorem C.5 (Worst-case Space Complexity of Parallel Sudoku Solver). *Let $S_{\mathcal{P}}(P_{\max})$ be the peak number of SAT calls that coexist in memory during the execution of strategy \mathcal{P} when the hardware can run at most P_{\max} calls concurrently. Then*

$$S_{\mathcal{P}}(P_{\max}) = \min\{B_{\text{cell}}, P_{\max}\}, \quad B_{\text{cell}} = (n^2)^{n^4}.$$

Proof. We again give matching upper and lower bounds.

Upper bound. By definition the execution platform schedules at most P_{\max} tasks simultaneously; therefore the solver can never hold more than P_{\max} tasks at once. If the total branch count B_{cell} is itself smaller than this cap, the very first batch contains all B_{cell} tasks and no more. Hence the peak space is bounded above by $\min\{B_{\text{cell}}, P_{\max}\}$.

Lower bound. We distinguish two cases.

1. $B_{\text{cell}} \leq P_{\max}$. All tasks fit into a single batch; the solver must keep B_{cell} tasks simultaneously.
2. $B_{\text{cell}} > P_{\max}$. By the hardware limit, every batch is filled to capacity with P_{\max} distinct tasks. Thus, there exists at least one instant at which the solver stores exactly P_{\max} concurrent tasks.

Tightness. The upper bound $\min\{B_{\text{cell}}, P_{\max}\}$ is met in both cases above, giving the matching lower bound and establishing

$$S_{\mathcal{P}}(P_{\max}) = \min\{B_{\text{cell}}, P_{\max}\}.$$

□

Therefore, even the simplest form of parallel execution can significantly enhance time efficiency in solving this complex task with a fixed space complexity by fully leveraging parallelism.

D Prompt for Data Curation

In this section, we release our five-stage prompting protocol to create Multiverse-1K, powered by Gemini 2.5 Pro. This protocol is designed to transform any sequential CoT data into Multiverse data.

This protocol starts with a multi-round conversation with the LLM (Stages 1-3) to convert an original reasoning chain into a parallel-structured summary. Subsequently, in Stage 4, both this summary and the full original reasoning trajectory are fed to the LLM to repopulate each summarized step with its complete, original details. A content checker then immediately assesses these refilled steps. If the *editor distance* (e.g., Levenshtein distance between the original trajectory (s_{ori}) and its rewritten version (s_{gen}), denoted as $d(s_{\text{ori}}, s_{\text{gen}})$) is too high, that step is re-generated. To normalize this, a *relative editor distance* is calculated to determine if a threshold r is exceeded (set to 0.2 in practice):

$$\text{Relative Editor Distance} = \frac{d(s_{\text{ori}}, s_{\text{gen}})}{\max(\text{length}(s_{\text{ori}}), \text{length}(s_{\text{gen}}))}$$

Next, in the first part of Stage 5, the output from Stage 4 is transformed into a MapReduce-structured reasoning trajectory by inserting the Map and Reduce phases that is generated by the LLM. To ensure the structural validity of the data, we perform a grammar check using a customized XML interpreter, which filters out invalid entries and extracts the outermost MapReduce blocks in the remaining valid ones. Subsequently, in the second part of Stage 5, each path within these blocks is rewritten separately to produce fully independent reasoning paths. The prompts used in the entire protocol are as follows:

STAGE 1: Generating a Summary Tree

Main-Step Extraction

Analyze the given reasoning chain (for a math or coding problem) and pull out every **major** step. Ignore substeps—only list the top-level insights or actions.

Output format

S1: [First major step]
S2: [Second major step]

118

S3: [Third major step]
 ...
 SX: [Description of step X]
 ...

Guidelines

- Label each top-level step consecutively ('S1', 'S2', 'S3', ...).
- Please capture the entire thought process presented in the reasoning chain, and do not skip any step that includes but not is limited to:
 1. Initial problem understanding and analysis
 2. All exploration paths (both successful and unsuccessful)
 3. Case studies, checks, or tests performed
 4. Any "aha" or correction (re-evaluation or re-thinking) moments
 5. The final reasoning that yields the solution
- Keep each item concise yet descriptive.
- Do **not** include any sub-numbering (no 'S2.1', etc.).
- Explicitly split multiple cases or scenarios into different steps. Each case should be allocated an independent step.

Substep Extraction

Given the output including all main step from a reasoning chain, break it down into all its internal substeps only if it can be meaningfully subdivided into smaller thought units.

Output format

S1: [Description of step 1]
 S2: [Description of step 2]
 S2.1 [Description of step 2.1]
 S2.2 [Description of step 2.2]
 ...
 S2.10 [Description of step 2.10]
 S3: [Description of step 3]
 S4: [Description of step 4]
 ...
 S10: [Description of step 10]
 ...

Guidelines

- Use the same parent index ('x') as the main step (e.g. if breaking down 'S2', label 'S2.1', 'S2.2', ...).
- Capture the entire thought process presented in the reasoning chain, and do not skip any substep that includes but is not limited to:
 1. Initial problem understanding and analysis
 2. All exploration paths (both successful and unsuccessful)
 3. Case studies, checks, or tests performed
 4. Any "aha" or correction (re-evaluation or re-thinking) moments
 5. The final reasoning that yields the solution
- Do **not** introduce deeper nesting larger than 2 (e.g. 'S2.1.1' is not allowed).
- Explicitly split multiple cases or scenarios into different substeps. Each case should be allocated an independent substep.

119

STAGE 2: Identifying Parallel Groups

Parallelizing Main Steps

Using only the **main steps** (S1, S2, ...) you extracted in Stage 1, identify all steps or contiguous step groups that can be executed in parallel without violating logical dependencies, and rewrite the plan as a structured parallel execution outline.

1. Identify Parallel Groups

- Find sets of adjacent main steps with no dependencies among them.

120

- Label groups P1, P2, ... and list their step ranges (e.g. [S1+S2, S3], [S4]).
- 2. *Rewrite into a Parallel Execution Plan*
- Preserve each step's original wording as much as possible.

Output Format:

Parallel groups:

P1: [S1+S2, S3]

P2: [S4]

...

Parallel execution plan:

P1[parallel reason: ...]:

S1+S2: [text of S1 + text of S2]

S3: [text of S3]

P2[parallel reason: ...]:

S4: [text of S4]

...

Guidelines

- **Coverage:** Include **every** step exactly once, either alone or inside a parallel group.
- **Contiguous Blocks:** Combine only adjacent steps into blocks; do **not** combine non-adjacent steps.
- **Strict Parallelism Only:** Build a dependency graph: draw an edge from step A to B if B uses A's output. A group P_i may include steps (or blocks) only if there are no edges between them. Treat conditional branches as independent tasks.
- **Contiguous Grouping Only:** Each parallel group must cover a continuous sequence of steps. Do not parallelize non-adjacent steps.
- **Conciseness:** Keep each bullet short and stick closely to the original text.

Parallelizing Substeps

Using only the **substeps** (S2.1, S2.2, ...) you extracted in Stage 1, identify all substeps or contiguous substep groups can be executed in parallel without violating logical dependencies, and rewrite the plan as a structured parallel execution outline.

1. Identify Parallel Groups

- Find sets of adjacent main steps with no dependencies among them.
- Label groups P1, P2, ... and list their step ranges (e.g. [S2.1+S2.2, S2.3], [S3.1]).

2. Rewrite into a Parallel Execution Plan

- Preserve each step's original wording as much as possible.

Output Format:

Parallel groups:

P1: [S2.1+S2.2, S2.3]

P2: [S2.4]

P2: [S3.1]

...

Parallel execution plan:

P1[parallel reason: ...]:

S2.1+S2.2: [text of S2.1 + text of S2.2]

S2.3: [text of S2.3]

P2[parallel reason: ...]:

S3.1: [text of S3.1]

...

Guidelines

- **Coverage:** Include **every** substep exactly once, either alone or inside a parallel group.
- **Contiguous Blocks:** Combine only adjacent substeps into blocks; do **not** combine non-adjacent substeps.
- **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 P_i may include steps (or contiguous

blocks) only if there are no edges between any two 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.

- **Contiguous Grouping Only:** Each parallel group must cover a continuous sequence of steps or blocks. In other words, you may only parallelize adjacent substeps. The occurrence of substeps in parallel groups must follow their original order. For example, P1: [S2.2, S3.1] is not allowed.
- **Conciseness:** Keep each bullet short and stick closely to the original text.

122

STAGE 3: Reformating into Parallel Structures

Get Structured Summary

Please summarize the conversation above by extracting the reasoning steps and substeps in Stage 1 as a tree structure with explicit parallelism annotations following Stage 2.

Output Format

```
O1: [Brief summary of top-level step S1]
<parallel>[parallel reason: ...]
O1.1: [Summary of substep S1.1]
O1.2: [Summary of substep S1.2]
...
</parallel>
<parallel>[parallel reason: ...]
O2: [Brief description of top-level step S2]
<parallel>[parallel reason: ...]
O2.1: [Summary of substep S2.1 + Summary of substep S2.2]
O2.2: [Summary of substep S2.3]
...
</parallel>
O3: [Brief description of top-level step S3]
</parallel>
O4: [Brief description of top-level step S4]
...
```

Guidelines

- **Max depth of nested <parallel> is 2.** Do not nest <parallel> tags more deeply than two levels.
- **Max depth of nested numbering is 2.** Only use 0x and 0x.y; do not introduce deeper numbering like 0x.y.z.
- **Sequential subpaths stay unexpanded.** If a node's children are purely sequential, list them normally without any <parallel> wrapper.
- **Tag parallel blocks.** Wrap only genuinely parallelizable sibling steps in a <parallel>...</parallel> block, and include a parallel-reason annotation.
- **Concise summaries.** Each step and substep should be described briefly and clearly.
- **Avoid over-splitting.** If most children are sequential and only a pair can run in parallel, either leave the group un-split or tag only the truly parallel pair.
- **Group parallelizable sets.** You may combine several independent paths into one <parallel> block when they share no dependencies.

123

STAGE 4: Refilling Original Details

Refill the Full, Detailed Reasoning Trajectories into the Structured Summary

You will receive an outline that *may be incomplete but includes* <parallel> tags indicating parallel structures. It contains summaries for several steps and substeps. You will also receive the corresponding original text, where sentences implicitly or explicitly map to hierarchical prefixes (e.g., O1, O1.1, O2) in sequence. Your task is to process the original reasoning chain sequentially to update the outline: replace existing summaries or insert new steps as needed, while preserving the original <parallel> tag structure.

Guidelines:

- **Initialize Structure** Start with the structure provided by the input outline, including its text/summaries and all <parallel> tags in their original locations.

124

- **Read Sentences Sequentially:** Process each sentence of the original text one by one, in the exact order they appear.
- **Process Each Sentence:**
 1. Determine the hierarchical prefix associated with this sentence (e.g., 01, 01.1, 02).
 2. Check if a step or substep with this prefix already exists in the outline.
 3. *If it exists:* Replace its current summary with the full original sentence.
 4. *If it does not exist:* Insert a new step/substep at the correct hierarchical position (e.g., S1.1 under S1, S2 after S1), using the full original sentence as its content and matching the outline's indentation.
- **Preserve <parallel> Tags:** Keep every existing <parallel> and </parallel> tag exactly where it was in the input outline. Do not add, remove, or relocate any tags.
- **Ensure Correct Output Formatting:**
 - Maintain proper hierarchical indentation for all steps and substeps.
 - Each entry must be on its own line, beginning with its prefix (e.g., 01:, 01.1:), followed by the full original sentence.
- **Maintain Completeness:** Verify that every sentence from the original reasoning chain has been processed and appears in the updated outline. Do not omit or merge any sentences.

125

STAGE 5: Adding MapReduce Structures & Rewriting All Paths

Filling Detailed Goal and Conclusion Based on the New Reasoning Trajectory

Based on the generated reasoning chain, your task is to transform it according to the following rules:

Output Format

```
[Full reasoning copied from the reasoning chain for the first top-level path]
[Full reasoning copied from the reasoning chain for the second top-level path]
...
Let's think in parallel.
<Parallel>
<Goal>
Path: [brief, self-contained description of case A]
Path: [brief, self-contained description of case B]
...
</Goal>
<Path>
[Introductory reasoning for case A]
Let's think in parallel.
<Parallel>
<Goal>
Path: [brief, self-contained description of case A.1]
Path: [brief, self-contained description of case A.2]
</Goal>
<Path>
[Full detailed reasoning for case A.1, rewritten clearly and independently]
</Path>
<Path>
[Full detailed reasoning for case A.2, rewritten clearly and independently]
</Path>
<Conclusion>
[Your concise summary of outcomes from A.1 and A.2]
</Conclusion>
</Parallel>
</Path>
<Path>
[Full detailed reasoning for case B, rewritten clearly and independently]
</Path>
...
<Conclusion>
[Your concise summary of outcomes from A and B]
</Conclusion>
</Parallel>
[Full detailed reasoning for any remaining paths]
```

126

Guidelines

- Remove all numbering labels (e.g., 01, 02.1) and eliminate any indentation.
- For each `<Parallel> . . . </Parallel>` block:
 - Group every step, substep, and subsubstep belonging to the same parallel branch into a single `<Path> . . . </Path>` section.
 - Discard the `[parallel reason: . . .]` annotations.
- Within each `<Parallel>` block:
 - Insert `<Goal>` before the first `<Path>`, listing each branch as `Path: . . .`
 - Insert `<Conclusion>` after the last `<Path>`, summarizing each branch's outcome independently.
- When multiple `<Path>` entries stem from the same original sentence or have interdependencies:
 - Rewrite each path separately and completely, ensuring no cross-references.
 - Provide enough context in each `<Path>` so it stands alone.
 - Fully encapsulate the logical reasoning for each path.
- Avoid repetition: do not echo the brief descriptions from `<Goal>` inside the corresponding `<Path>`, and minimize redundant information across paths.

Rewriting Paths in the Structured Reasoning Trajectory

You are given a full structured reasoning trajectory inside a `<Parallel>` block, consisting of:

- one `<Goal>` block with multiple `<Outline>` elements
- multiple `<Path>` blocks
- one `<Conclusion>` block.

Some `<Path>` blocks may contain an entire nested `<Parallel>` structure (from `<Parallel>` to `</Parallel>`). These nested blocks should be rewritten using the same rules recursively.

For `<Goal>`:

- Rewrite each `<Outline>` into a **concise statement of what is being calculated or determined**.
- Remove any content describing **how** the problem is solved or intermediate reasoning steps.

For each `<Path>`:

- Keep the original numbering prefix (e.g., '1:', '2:').
- Rewrite the content as a **complete, fluent, and logically self-contained paragraph**.
- Do **not** use transitional phrases like "First," "Then," "Next," "On the other hand," etc.
- If the `<Path>` contains **five or fewer sentences**, rewrite them together as a single coherent paragraph, ensuring logical flow and fluency without using transitional phrases.
- If the `<Path>` contains **more than five sentences**: Rewrite the first five sentences together as a single unit, forming a fluent paragraph. For the remaining sentences, rewrite each one individually, based on its meaning, as clear and fluent standalone statements.
- If the `<Path>` contains a **nested `<Parallel>` block**, apply all these rules recursively to the nested block.

Each `<Path>` must make sense independently, even if it contains a nested reasoning chain.

For `<Conclusion>`:

- Rewrite the conclusion as the **most concise and synthesized summary** of the main outcomes from all `<Path>` blocks.
- You may combine or compare results from different paths, but keep it succinct and direct.

Nested `<Parallel>`:

- A nested `<Parallel>` may appear only **as a full block inside a `<Path>`**.
- If a `<Path>` contains a nested `<Parallel> . . . </Parallel>` block, process that inner block exactly as you would the top-level one:
 - Rewrite the inner `<Goal>`, `<Path>`, and `<Conclusion>` elements accordingly.
 - Maintain the XML structure — do not reindent or alter the tag hierarchy.

Output Format

```
<Parallel>
<Goal>
<Outline>
1: [concise description of the goal of Path 1]
```

```

</Outline>
<Outline>
2: [concise description of the goal of Path 2]
</Outline>
</Goal>
<Path>
1: [self-contained paragraph for Path 1, rewritten sentence by sentence]
<Parallel>
<Goal>
<Outline>
1.1: [concise goal for subpath 1.1]
</Outline>
<Outline>
1.2: [concise goal for subpath 1.2]
</Outline>
</Goal>
<Path>
1.1: [rewritten paragraph for subpath 1.1]
</Path>
<Path>
1.2: [rewritten paragraph for subpath 1.2]
</Path>
<Conclusion>
[summarized results from subpaths 1.1 and 1.2]
</Conclusion>
</Parallel>
</Path>
<Path>
2: [self-contained paragraph for Path 2, rewritten sentence by sentence]
</Path>
<Conclusion>
[summary of outcomes from Path 1 and Path 2]
</Conclusion>
</Parallel>

```

Guidelines

- You must preserve all XML tags exactly as they appear — do not remove, rename, or reorder tags.
- Only modify the inner text of <Outline>, <Path>, and <Conclusion> blocks.
- Wrap the final output in a single Markdown code block using triple backticks.

128

129 E Examples in Training Data and Model Generation

130 We present several examples from our Multiverse-1K dataset, along with the corresponding outputs
 131 generated by Multiverse-32B, which was trained on this dataset.

132 E.1 Training Examples from Multiverse-1K

Multiverse Data A: Selective Case

```

<Parallel>
<Goal>
<Outline>
1: Determine if  $H$  can be constructed as a countable intersection of open dense sets using Baire Category Theorem (BCT), leveraging the completeness of  $S^2$ .
</Outline>
<Outline>
2: Determine if  $H$  can be constructed step-by-step using  $\Omega$ 's approximation property to ensure density and  $K$ -avoidance for chords, while addressing potential difficulties.
</Outline>
</Goal>

```

133

<Path>

1: The sphere S^2 is a complete metric space, suggesting the possibility of constructing H as a countable intersection of open dense sets; however, the specific method for such a construction is not immediately clear.

</Path>

<Path>

2: A step-by-step construction of H could leverage the given condition to ensure density and that all chords avoid K ; H must be dense in S^2 , meaning it has points in every open subset of S^2 , and any two points in H must have their connecting chord disjoint from K . The family Ω allows approximation of any two points on S^2 with a chord $X'Y'$ from Ω . This implies Ω is dense in the space of all chords, with the additional property that its chords avoid K . The closed nature of K means its complement within the unit ball is open, so a chord disjoint from K lies entirely within this open complement. Consequently, Ω comprises chords in the complement of K . The challenge is to construct a dense set H such that all chords connecting points within H avoid K , given that for any $X, Y \in S^2$, nearby points X', Y' exist whose chord $X'Y'$ is in Ω and thus avoids K . One approach might be to construct H as a countable dense set where each pair's connecting chord belongs to Ω . Achieving density alone is insufficient; all chords between any two points in H must be in Ω . Directly constructing such an H presents significant difficulties.

</Path>

<Conclusion>

The Baire Category Theorem approach for constructing H via intersections (Path 1) lacks a clear method. A step-by-step construction (Path 2) highlights the requirements of density and K -avoidance for all chords within H , but faces challenges in maintaining the chord property universally during the construction process.

</Conclusion>

</Parallel>

134

Multiverse Data B: Collective Case

<Parallel>

<Goal>

<Outline>

1: Determine the sum of distances from P_1 to P_2 , P_3 , and P_4 .

</Outline>

<Outline>

2: Determine the sum of distances from P_2 to P_1 , P_3 , and P_4 .

</Outline>

<Outline>

3: Determine the sum of distances from P_3 to P_1 , P_2 , and P_4 .

</Outline>

<Outline>

4: Determine the sum of distances from P_4 to P_1 , P_2 , and P_3 .

</Outline>

</Goal>

<Path>

1: For point $P_1(0, 0)$, the distance to $P_2(10, 20)$ is

$$\sqrt{(10 - 0)^2 + (20 - 0)^2} = \sqrt{100 + 400} = \sqrt{500} \approx 22.36.$$

The distance to $P_3(5, 15)$ is

$$\sqrt{(5 - 0)^2 + (15 - 0)^2} = \sqrt{25 + 225} = \sqrt{250} \approx 15.81.$$

The distance to $P_4(12, -6)$ is

$$\sqrt{(12 - 0)^2 + (-6 - 0)^2} = \sqrt{144 + 36} = \sqrt{180} \approx 13.42.$$

The sum is $22.36 + 15.81 + 13.42 \approx 51.59$.

</Path>

<Path>

2: For point $P_2(10, 20)$, the distance to $P_1(0, 0)$ is

$$\sqrt{(10 - 0)^2 + (20 - 0)^2} = \sqrt{500} \approx 22.36.$$

The distance to $P_3(5, 15)$ is

$$\sqrt{(10 - 5)^2 + (20 - 15)^2} = \sqrt{25 + 25} = \sqrt{50} \approx 7.07.$$

135

The distance to $P_4(12, -6)$ is

$$\sqrt{(10-12)^2 + (20-(-6))^2} = \sqrt{4+676} = \sqrt{680} \approx 26.08.$$

The sum is $22.36 + 7.07 + 26.08 \approx 55.51$.

</Path>

<Path>

3: For point $P_3(5, 15)$, the distance to $P_1(0, 0)$ is

$$\sqrt{(5-0)^2 + (15-0)^2} = \sqrt{250} \approx 15.81.$$

The distance to $P_2(10, 20)$ is

$$\sqrt{(5-10)^2 + (15-20)^2} = \sqrt{50} \approx 7.07.$$

The distance to $P_4(12, -6)$ is

$$\sqrt{(5-12)^2 + (15-(-6))^2} = \sqrt{49+441} = \sqrt{490} \approx 22.14.$$

The sum is $15.81 + 7.07 + 22.14 \approx 45.02$.

</Path>

<Path>

4: For point $P_4(12, -6)$, the distance to $P_1(0, 0)$ is

$$\sqrt{(12-0)^2 + (-6-0)^2} = \sqrt{180} \approx 13.42.$$

The distance to $P_2(10, 20)$ is

$$\sqrt{(12-10)^2 + (-6-20)^2} = \sqrt{4+676} = \sqrt{680} \approx 26.08.$$

The distance to $P_3(5, 15)$ is

$$\sqrt{(12-5)^2 + (-6-15)^2} = \sqrt{49+441} = \sqrt{490} \approx 22.14.$$

The sum is $13.42 + 26.08 + 22.14 \approx 61.64$.

</Path>

<Conclusion>

The sums of distances from P_1, P_2, P_3, P_4 to the other three points are approximately 51.59, 55.51, 45.02, and 61.64, respectively.

</Conclusion>

</Parallel>

136

137 E.2 Generated Examples from Multiverse-32B

Multiverse Generation A: Selective Case

<Parallel>

<Goal>

<Outline>

1: Count horizontal vs. vertical segments.

</Outline>

<Outline>

2: List possible H/V patterns.

</Outline>

</Goal>

<Path>

1: A path from $(0, 0)$ to $(8, 8)$ uses exactly eight right moves (R) and eight up moves (U). With five segments alternating direction, starting with R forces the pattern

$$R - U - R - U - R,$$

giving three horizontal segments and two vertical segments. Conversely, starting with U yields

$$U - R - U - R - U,$$

for three vertical and two horizontal segments. In either case there are four direction changes (one at each segment boundary), and the segment counts differ by one because there are an odd number of segments.

</Path>

<Path>

138

2: The total move count is $16(8R + 8U)$. If $H = 3$ and $V = 2$, then the three horizontal segment lengths $h_1 + h_2 + h_3 = 8$ with each $h_i \geq 1$, which has

$$\binom{8-1}{3-1} = \binom{7}{2} = 21$$

compositions. The two vertical segments $v_1 + v_2 = 8$ with $v_j \geq 1$ has

$$\binom{8-1}{2-1} = \binom{7}{1} = 7$$

compositions, for $21 \times 7 = 147$ ways when starting horizontal. Similarly, if $V = 3$ and $H = 2$, there are 21 ways to split 8 into three vertical segments and 7 ways for two horizontal segments, again 147 ways. In total there are $147 + 147 = 294$ segment-length configurations consistent with five alternating segments summing to $(8, 8)$.

</Path>

<Conclusion>

A five-segment path must alternate directions, yielding either $(H, V) = (3, 2)$ or $(2, 3)$ segments. There are 147 ways to allocate segment lengths in each case, for 294 total valid patterns.

</Conclusion>

</Parallel>

139

Multiverse Generation B: Collective Case

<Parallel>

<Goal>

<Outline>

1: Compute Case 1 count.

</Outline>

<Outline>

2: Compute Case 2 count.

</Outline>

</Goal>

<Path>

1: In Case 1 the path pattern is H-V-H-V-H, giving three horizontal and two vertical segments. We distribute 8 units into three positive horizontal segments:

$$\binom{8-1}{3-1} = \binom{7}{2} = 21.$$

We distribute 8 units into two positive vertical segments:

$$\binom{8-1}{2-1} = \binom{7}{1} = 7.$$

Total for Case 1:

$$21 \times 7 = 147.$$

</Path>

<Path>

2: In Case 2 the pattern is V-H-V-H-V, giving three vertical and two horizontal segments. Distribute 8 units into three positive vertical segments:

$$\binom{7}{2} = 21.$$

Distribute 8 units into two positive horizontal segments:

$$\binom{7}{1} = 7.$$

Total for Case 2:

$$21 \times 7 = 147.$$

</Path>

<Conclusion>

Case 1 yields 147 paths; Case 2 yields 147 paths; overall $147 + 147 = 294$.

</Conclusion>

</Parallel>

140

141 F Full Results of the Efficiency Analysis

142 In Table 1, we provide the full results of our efficiency analysis with varying batch sizes. Since our
 143 Multiverse does not allow explicit control over the degree of parallelism, we run inference with a
 144 fixed generation length and measure both the observed degree of parallelism and the corresponding
 145 inference speedup. To evaluate performance at a specific degree of parallelism P , we average results
 146 from samples whose degree of parallelism falls within a window of ± 0.05 around P . For instance, to
 147 plot the curve for $P = 1.1$, we include samples with degree of parallelism between 1.05 and 1.15.

Table 1: Full Speedup Results Across Varying Degrees of Parallelism and Batch Sizes

# Parallel	bsz=4	bsz=8	bsz=16	bsz=32	bsz=64	bsz=128
1.1	1.07 ± 0.02	1.13 ± 0.02	1.06 ± 0.01	1.07 ± 0.01	1.08 ± 0.02	1.15 ± 0.02
1.2	1.20 ± 0.02	1.18 ± 0.02	1.18 ± 0.01	1.18 ± 0.02	1.19 ± 0.04	1.22 ± 0.02
1.3	1.31 ± 0.02	1.25 ± 0.03	1.32 ± 0.02	1.31 ± 0.01	1.31 ± 0.02	1.35 ± 0.02
1.5	1.47 ± 0.03	1.46 ± 0.02	1.54 ± 0.02	1.51 ± 0.02	1.50 ± 0.01	1.55 ± 0.03