DiTTo: Distributed Test-Time Compute Scaling

Raj Sangani, Isita Bagayatkar, Gnana Heemmanshuu Dasari

CS 230 Winter 2025 — Group 6

1 Introduction

LLMs are being applied to multitudes of applications – from personal daily usage to complicated, automated workflows deployed by companies. This requires LLMs to generate better outputs using chain-of-thought reasoning – as a person would go about reasoning – all the while meeting low latency constraints.

Recent work shows that exploring multiple generation paths while sampling from language models can greatly improve a model's ability at tasks that require "reasoning." Chain-of-thought reasoning is achieved through multi-step LLM generations. This benefits from exploring multiple, diverse generation paths. The best generation paths are built on, leading to a better answer than just serially generating one chain-of-thought output.

1.1 Motivation

Training large models requires lots of storage for data and computing for training. Work in the ML model training space leverages distributed systems to distribute heavy computation. LLMs notoriously need gigabytes of storage, data, and computing for training and benefit from distributed training and inferencing.

Additionally, exploring multiple generation paths can be expensive in terms of generation time and token cost. We design a distributed system DiTTo that can iteratively search for the best generations and rank them using multiple nodes and thus reduce the time taken to generate an answer. It is also no secret that a model's size is proportional to its reasoning capabilities. We will evaluate how a small model equipped with extensive exploration during inference compares to a large model's performance – on a dataset of mathematical reasoning-based queries.

1.2 Contributions

Our contributions include:

1. Distributed system for test-time compute scaling
2. Speed-up (up to 4x) and throughput analysis
3. Discussion of design tradeoffs

2 Background

2.1 Test-Time Compute Scaling

[Snell et al.(2024)] introduced test-time compute scaling via multipath exploration. It achieves 4x efficiency and can outperform a 14x larger model. Given a query, multiple LLM generation paths are explored. Each generation is different. At the end of each round, the generations are scored and pruned using the Process Reward Model (PRM) module. A complete answer requires k rounds of generations. Let's call intermediate queries and answers subqueries and subanswers. A subquery is sent out, multiple generations are ranked and pruned via PRM, and the subanswer is concatenated to the original query in subsequent rounds. Figure 1 shows an overview of test-time compute scaling.

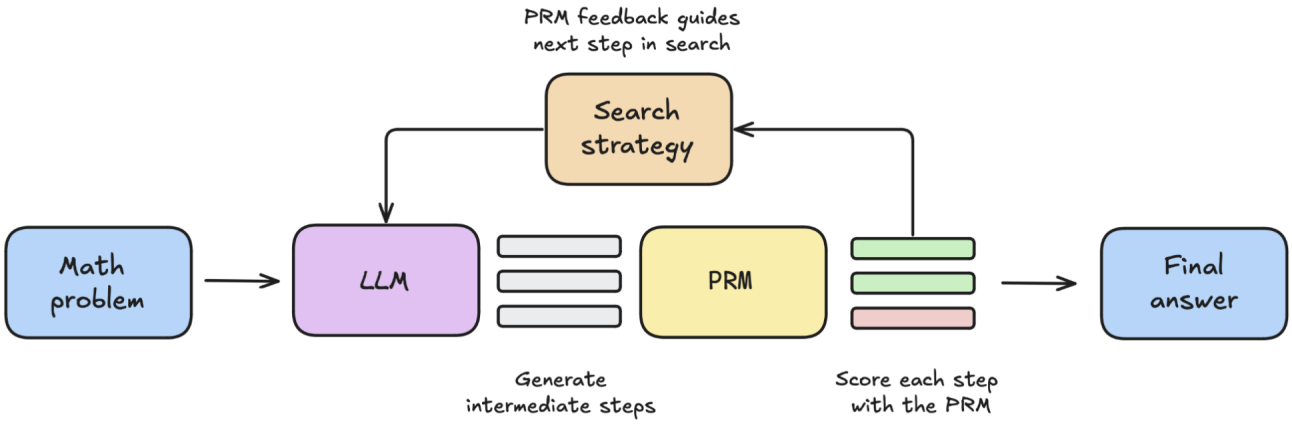

Figure 1: foo

Furthermore, the PRM module can use a variety of ranking and pruning methodologies, such as Best-of-N, Beam Search, and Diverse Verifier Tree Search (shown in Figure 2).

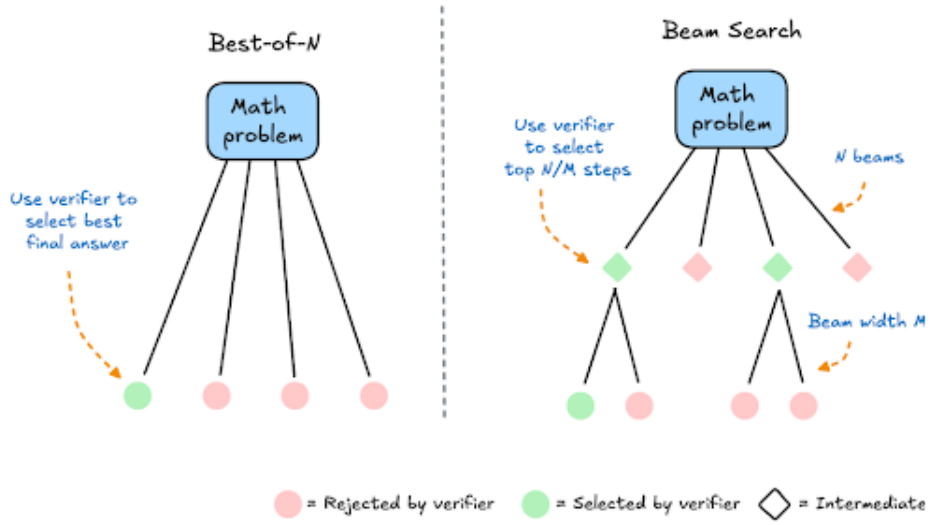

Figure 2: PRMs

2.2 Model Size and Reasoning Capabilities

As evidenced in LLM scaling law [Kaplan et al.(2020)], LLM model size is positively correlated with its inference performance. Chain-of-thought reasoning is used to achieve human like thinking using LLMs and to check the model's reasoning for soundness. It also serves to prevent hallucinations. Work in this space uses open-source ChatGPT logs to observe how users prompt LLM models (demonstrating a chain-of-thought) to go from their initial question to receiving an acceptable answer from the LLM [Wei et al.(2023), Zhao et al.(2024)].

Larger models are being developed and becoming increasingly available; however, it is possible to achieve the performance of larger models using smaller models by exploring multiple generation paths [Snell et al.(2024)].

This is useful in terms of using cheaper, smaller models, but also in deployment scenarios where these models are deployed locally in a compute-restrained manner.

2.3 Distributed Model Training

ML models are trained in a distributed manner to distribute storage and compute costs. LLMs require gigabytes of storage and compute. LLM fine-tuning – making an LLM more specialized to a dataset of domain knowledge – can be done by updating all parameters or using LoRA or QLoRA, where most parameters are frozen and the top parameters are finetuned to the dataset [Hu et al.(2021), Dettmers et al.(2023)]. Distributed LLM training is an emerging field [Dong et al.(2025)].

Using a centralized approach, individual nodes get data, train on that data, and the updated weights are sent to the centralized server to be aggregated. The centralized server uses an averaging algorithm to aggregate weights. The new model is re-broadcasted out to the edge nodes, so they can retrain. Of course, distributed systems add communication latency between the aggregator node and the worker nodes. This introduces a latency+node serving cost versus compute/storage costs tradeoff. [Huang et al.(2024)] is a framework for secure distributed LLM training that leverages model slicing and trusted execution environments. [Tang et al.(2024)] also supports load-balancing given geographically distributed GPUs, handles heterogeneous software and hardware, and adaptive compression at slow communication links. [Wang et al.(2024)] introduces micro-execution batching as a way to address communication latency in distributed training.

Tangentially, federated learning also leverages distributed training to train models in a privacy-preserving manner. Here the data never leaves the nodes where the data is collected. Models are trained on the data locally, the weights differences are sent to the central model, and an algorithm is used to aggregate all the weights and update the model. This new model is broadcast out to the nodes for inference. This, however, has to handle the heterogeneity of data at different nodes while aggregating weights at the central server.

Distributed inferencing (test-time compute) parallelizes inferencing.

3 System Design Overview

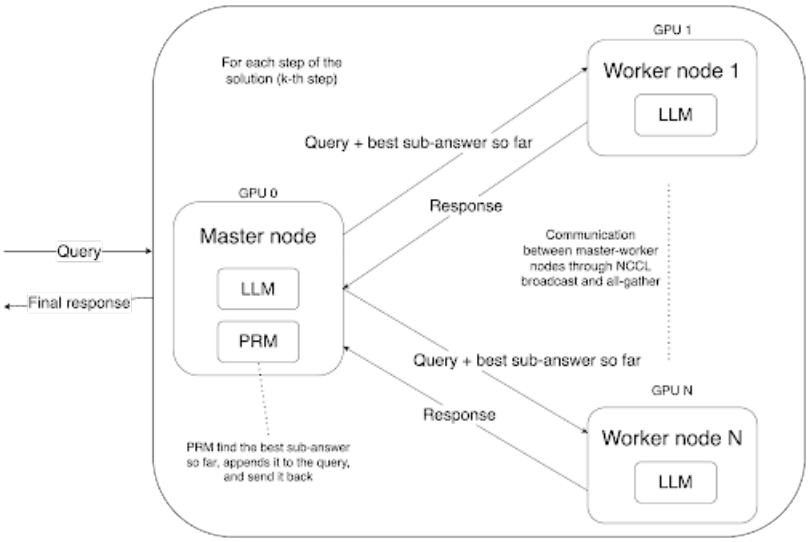

Figure 3: System Architecture

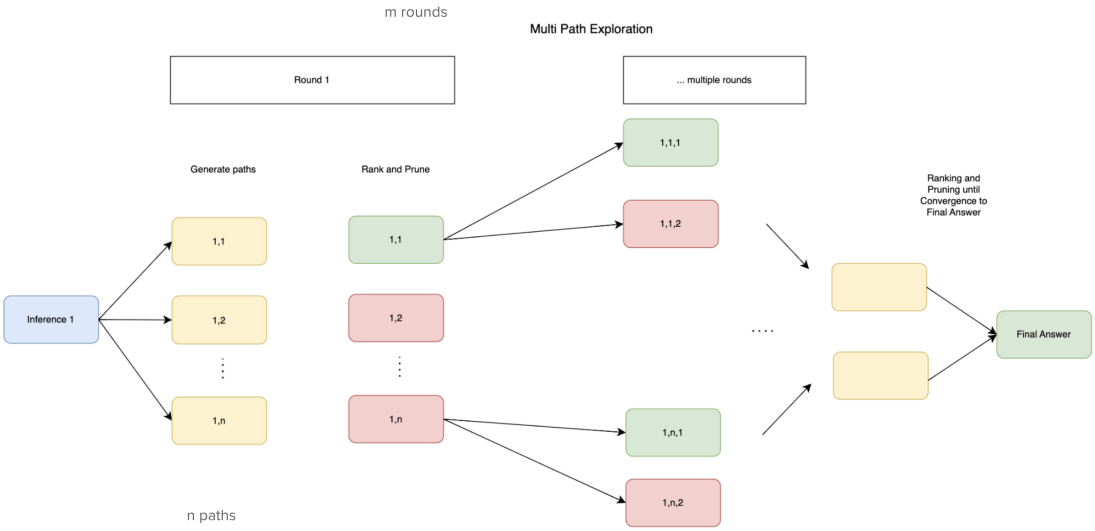

Figure 4: DiTTo Test-Time using Best-of-N

3.1 Overview

3.1.1 Provisioning

This system is comprised of (a) a master node and (b) N GPU worker nodes. The master node contains the LLM for inferencing and PRM. The N GPU nodes contain the LLM for inferencing. DiTTo PRM uses the Best-of-N pruning algorithm.

DiTTo explores predefined m exploration paths. A random seed is used to ensure the diversity of outputs on each exploration path. Note that the number of steps (rounds) taken by the query (i.e. number of inference steps in chain-of-thought reasoning) depends on the query. The number of rounds (k) needed by the query to converge to the final answer is determined before the k rounds are run. Let's say that a query takes k rounds.

3.1.2 Runtime

In the algorithm, the master node receives a query from a client. The master node **broadcasts** the query to m nodes to explore m exploration paths. The GPU nodes receive the query, have the LLM inference, and output a subanswer. Responses from GPUs are sent to and gathered by the master node. At the master node, the PRM scores all received outputs (subanswer), chooses the highest-ranked subanswer, and prunes the rest (m-1 generations). In consequent rounds, the query and subanswer are concatenated to form the new query; this is sent out to m exploration paths.

This process is repeated for k-1 rounds. At the end of k rounds, the master node has the answer to the query.

3.2 Multipath exploration

DiTTo explores a predefined m number of paths. Random seed ensures diverse generations so that the m paths do not all generate the same outputs. The scoring algorithm assesses the performance of the generated output. Best-of-N PRM is used to reduce the exponential multipath exploration resulting from Beam Search.

Note that each query requires a unique number of rounds. Therefore, the inputs to the system (queries) are heterogeneous. Figure 4 shows DiTTo Best-of-N PRM across multiple paths and rounds.

3.3 Maxmizing Channel Utilization

DiTTo maximizes GPU utilization by batching queries. The GPU is not being utilized when the query response is sent to master, master runs PRM, and sends out a new query with the subanswer. The naive approach to maximizing channel utilization would be maintaining task queues at each worker GPU node. Thus, worker nodes always have tasks in their queues. GPU utilization and task queue size can be used to handle load balancing. However, communication and PRM execution latency is less than the performance cost of maintaining CPU and GPU clad worker nodes, where the CPU maintains queues and communicates with the GPU.

DiTTo batching batches together queries that take the same number of rounds. Thus, instead of serially running queries, the latency is now the maximum inference time taken by one query in the batch as opposed to the sum of the time taken to run all queries.

3.4 Fault Tolerance and Monitoring

DiTTo handles fault tolerance at worker nodes through elastically having the remaining N-1 GPUs continue computing. Fault tolerance at the master node is achieved through a monitoring script that detects when the system is down; if detected to be down, it is brought back up. The outputs are checkpointed at every generation round. Thus, in the recovery process, the model recovers prior state by loading the saved checkpoints and resuming execution from there.

Furthermore, each GPU worker node, periodically sends out its GPU utilization metrics to the master node.

DiTTo fault tolerance handles failover while nodes are executing; if nodes fail when nodes are not executing distributed training, this centralized heartbeat protocol logs the failover. This monitoring system also grants insight into the effectiveness of batching in maximizing GPU utilization.

4 Implementation

We implement DiTTo using the PyTorch framework to ensure communication between GPUs, with one Master GPU and remaining worker GPUs, each of 50GB. torch.distributed.all_gather and torch.distributed.broadcast are used to collect candidate solutions from each node, and distribute the selected solution path among all the nodes. These functions use NCCL (N̈ickel̈), the NVIDIA Collective Communications Library as the backend, which is the best performing algorithm for this usecase and is optimized for multi-GPU communication.

Greedy Beam Search

The algorithm uses a new Greedy Beam Search algorithm, which is a faster alternative to the beam search algorithm used for ranking answers. The master node takes all the candidate solutions from the worker nodes and itself and uses the PRM to find the best performing solution so far. This solution is then provided as a query+subanswer to all the worker nodes again, using which they generate further steps and provide candidate solutions to the master node. This cycle keeps going until the ¡EOS¿ token is reached.

Query Batching

This system has several steps. Communication between master and worker nodes, time taken by the PRM for ranking, and time taken for LLM inference at each node. Of these, we found the model inference to be the performance bottleneck. To optimize this process, we perform query batching. LLMs can infer on multiple queries in parallel, with the time taken for inference being equal to the maximum time for inference among all queries, which is less than the *sum* of time taken for all queries, the case in one-by-one inference. Query batching is limited by the memory constraints of the GPUs and the communication overhead it adds. For our use case, we calculated a batch size of 4 to be optimal, ensuring maximal GPU utilization while not causing overload.

Avoiding pipelining

While performing query batching, we naturally considered pipelining queries, so that as soon as query 1s candidate solutions are sent by each node t the master, query 2 is started to get processed on that worker node while the master ranks the candidate solutions for query 1, as shown below.

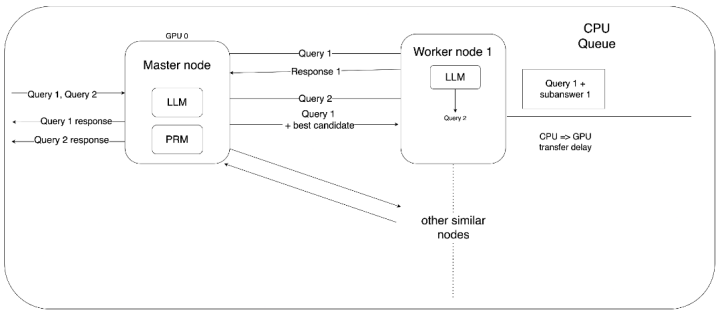

Figure 5: Pipelining introduces CPU-GPU data transfer delays

The problem with this approach arrives from two related reasons

1. PRM ranking is a very fast process, while LLM inference is the bottleneck. The gap is huge enough that it is guaranteed that query 1 results can be communicated, ranked and re-sent for

further processing by the Master while inference for query 2 is going on. This would require queuing.

2. Queues stored at the nodes for this reason would have to be stored in the CPU. Data transfer from CPU to GPU for inference processing takes a huge amount of time, with the performing worse off than it would be without pipelining.

Fault Tolerance

We added fault tolerance mechanisms for DiTTo with recovery mechanisms via checkpointing. After each step, DiTTo saves the state of the system in a *checkpoint.json* file, the overhead for this process is found to be small enough. In the case of a GPU failure, PyTorch detects a membership and restarts all the nodes. PyTorch elects a new Master randomly (which is sufficient given the design of DiTTo). This new Master node reads the state of the system from checkpoint.json and, if there is an ongoing solution that was paused, the candidate solutions till that point are picked up and processed further till a final answer is received.

5 Evaluation

Our primary goal with DiTTo is to improve the performance of test time compute scaling systems. Our initial experiments with HuggingFace's Search and Learn model itself showed that a serialized PyTorch implementation performs better than the vLLM one used by HuggingFace. We expect this to be due to PyTorch's improved capabilities for working with GPUs.

To analyze the performance of a distributed implementation of test time compute scaling with a serialized one, we compare the two corresponding implementations in PyTorch. This is also to allow for additional customizations to the scripts for improved testing. We evaluate DiTTo on the HuggingFaceH4/MATH-500 dataset, comprised of 500 problems from OpenAI's MATH reasoning dataset. This dataset also has solutions and information for grading model logic. We observe improvements in speed of up to **4x** with DiTTo, as seen in the execution time and speedup graphs below (Figure 6).

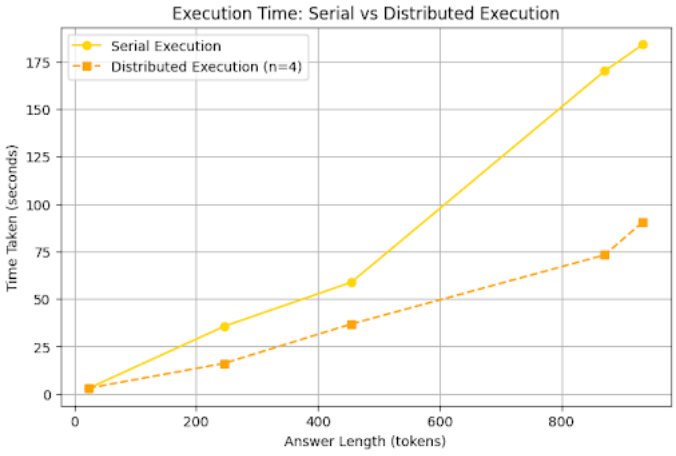

Figure 6: Execution time of a query executed with serial TTC vs distributed TTC (DiTTo)

Furthermore, recall that each round, the subanswer is appended to the query before it is sent out again. This makes the query longer and consequent inference time longer. The speedup is also

evaluated when the number of tokens is increased Figure 7. Here, we see an average speedup of 1.82x given 4 nodes.

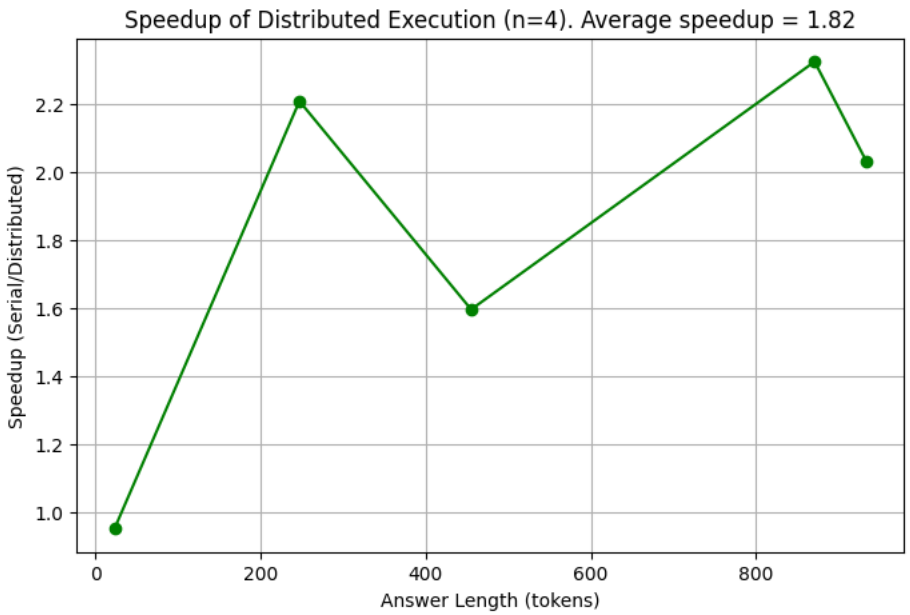

Figure 7: Performance Speedup given Answer Length

It was difficult to test the real-time fault tolerance of DiTTo due to restrictions on the GPU clusters available. The PyTorch library picks up failures only if a GPU is completely down, which would require sudo access. Instead, we induce a failure from a parallel terminal execution, manually restart the system with a lesser number of available GPUs and measure the time taken. As expected, the failure recovery mechanism with the checkpoint file allows the system to resume execution from where it stopped, taking only the time needed to complete the remaining steps.

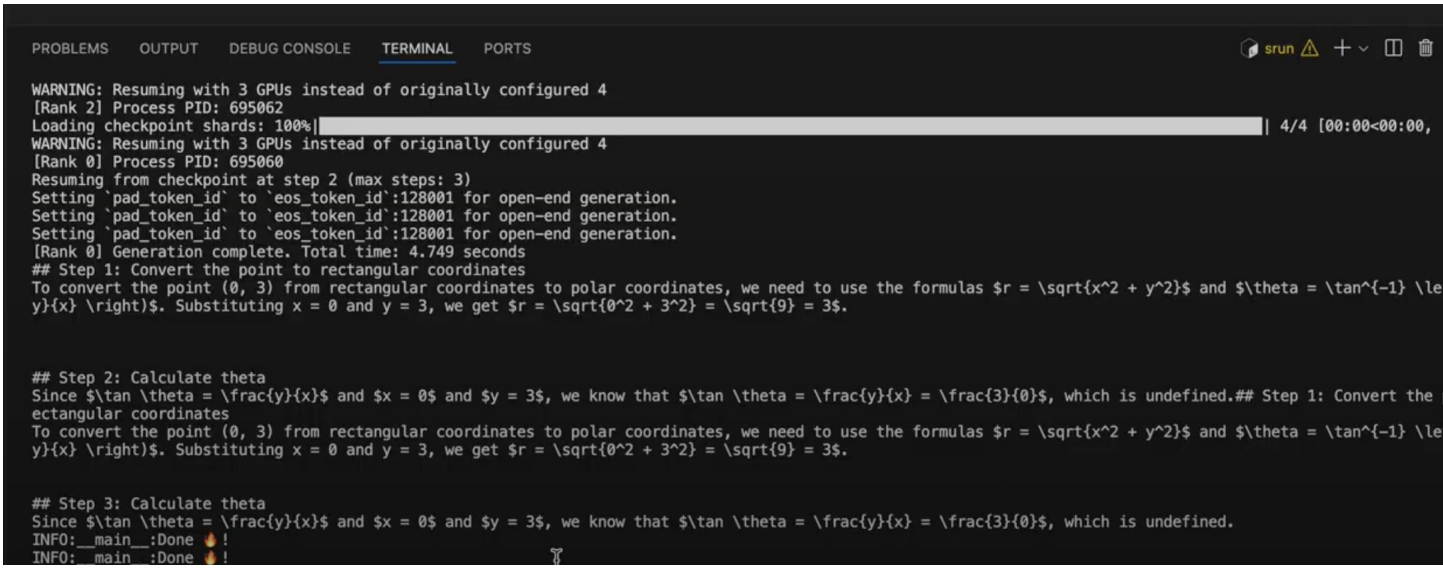

Figure 8: Fault Tolerance Mechanism adapts to less GPU worker nodes and resumes from checkpoint

It is to be noted that a manual restart would not be required in the wild; the PyTorch library detects GPU failures and restarts itself upon membership changes. Stopping the system from

restarting would require a low library level change.

We also tested DiTTo with a varying number of worker nodes to get a measure of the point where communication delays would be more significant than the inference bottleneck. As the graph below indicates, there is not much difference between our limits of 2/4/8 nodes.

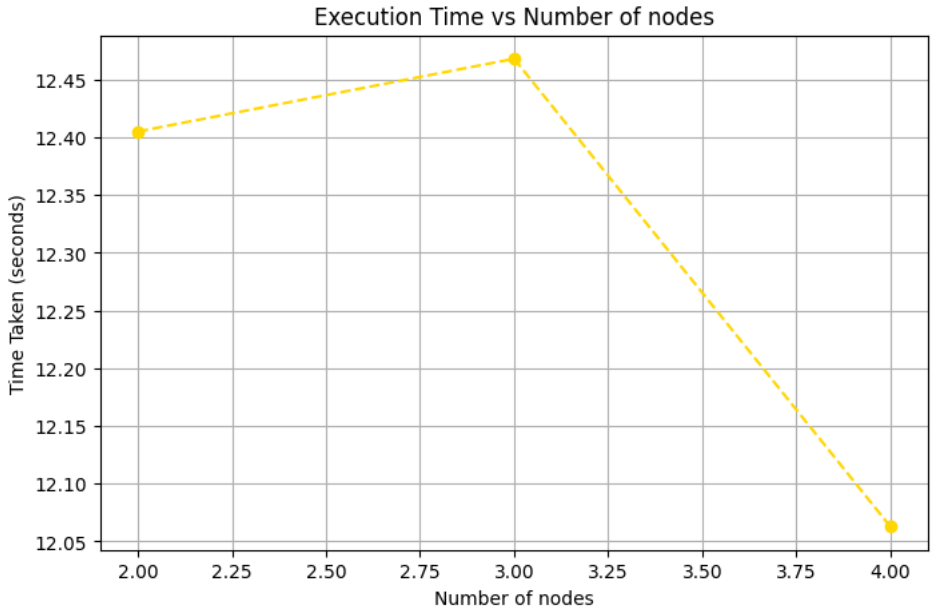

Figure 9: Throughput given different number of nodes

6 Discussion & Future Work

6.1 Path Exploration Design

In DiTTo, we execute each subquery round on m exploration paths. Note that once the query inferencing process converges towards the latter steps (i.e. closer to the final solution), the computation for mathematical problems does not diverge too much. Here, DiTTo performance can benefit from reducing the number of exploration paths in the latter rounds. Of course, for applications where the answer does not converge towards the end (i.e., more research like explorative queries), the number of exploration paths should not be reduced in latter rounds. Thus, depending on the DiTTo application, the nuxmber of exploration paths can be reduced in latter rounds. Future work would introduce a dynamic element that dynamically reduces multipath generations based on the task at hand and how much performance benefit can be yielded through multipath generation in latter steps. The goal here is to reduce unnecessary compute and increase overall system throughput.

6.2 PRM Alternatives

DiTTo performance can be further explored by analyzing the use of other PRM algorithms, such as beam search and diverse verifier tree search. Beam search generates exponentially more paths; thus, the pruning algorithm needs to be adapted to the application. This would require a dynamic algorithm that converges to a final solution while exploring the performance vs exponential paths tradeoff. Moreover, diverse verifier tree search ensures diverse path exploration; this is interesting for more "creative" generative tasks, such as story writing.

6.3 Heterogeneity of Queries

Queries require a different number of steps for inference depending on the complexity of the query. Also, in latter rounds – the query and subanswers get concatenated, building a longer query. Inferencing with this longer query takes longer. Thus, granted queries requiring different steps, we can either force it to use the minimum number of steps (as we do now), or assuming a system at scale, we can design a new batching algorithm that batches multitudes of these while meeting low latency requirements of the user.

Furthermore, DiTTo was evaluated on a math world problem dataset (and designed with math problems in mind). Usage of DiTTo can be for specific applications, such as a math tutoring service, or more general applications, like a chatbot interface that answers a variety of English and Math questions. Such heterogeneity would have to be supported using an algorithm that adapts the multipath exploration algorithm to how the input will adapt until convergence to the final answer.

Another element of heterogeneity would be to explore how the system performs under diverse versus homogeneous workloads. Relatively homogenous workloads would be doing the same math problems but with different numbers. Diverse workloads would be completely different types of math problems.

7 Conclusion

We introduce DiTTo, a distributed test time compute scaling system with significant applications in reasoning tasks. DiTTo outperforms the current serial execution systems with a speedup factor of 4x. It uses batched queries for faster outputs, and is fault tolerant. Future work can involve path exploration designs, multiple PRMs spread across worker nodes, and making the system even faster, with algorithms to batch similar queries together for better utilization.

References

[Dettmers et al.(2023)] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. arXiv:2305.14314 [cs.LG] https://arxiv.org/abs/2305.14314

[Dong et al.(2025)] Haotian Dong, Jingyan Jiang, Rongwei Lu, Jiajun Luo, Jiajun Song, Bowen Li, Ying Shen, and Zhi Wang. 2025. Beyond A Single AI Cluster: A Survey of Decentralized LLM Training. arXiv:2503.11023 [cs.DC] https://arxiv.org/abs/2503.11023

[Hu et al.(2021)] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL] https://arxiv.org/abs/2106.09685

[Huang et al.(2024)] Wei Huang, Yinggui Wang, Anda Cheng, Aihui Zhou, Chaofan Yu, and Lei Wang. 2024. A Fast, Performant, Secure Distributed Training Framework For LLM. In *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 4800–4804. https://doi.org/10.1109/ICASSP48485.2024.10446717

[Kaplan et al.(2020)] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG] https://arxiv.org/abs/2001.08361

[Snell et al.(2024)] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters. arXiv:2408.03314 [cs.LG] https://arxiv.org/abs/2408.03314

[Tang et al.(2024)] Zhenheng Tang, Xueze Kang, Yiming Yin, Xinglin Pan, Yuxin Wang, Xin He, Qiang Wang, Rongfei Zeng, Kaiyong Zhao, Shaohuai Shi, Amelie Chi Zhou, Bo Li, Bingsheng He, and Xiaowen Chu. 2024. FusionLLM: A Decentralized LLM Training System on Geo-distributed GPUs with Adaptive Compression. arXiv:2410.12707 [cs.DC] https://arxiv.org/abs/2410.12707

[Wang et al.(2024)] Haiquan Wang, Chaoyi Ruan, Jia He, Jiaqi Ruan, Chengjie Tang, Xiaosong Ma, and Cheng Li. 2024. Hiding Communication Cost in Distributed LLM Training via Micro-batch Co-execution. arXiv:2411.15871 [cs.DC] https://arxiv.org/abs/2411.15871

[Wei et al.(2023)] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

[Zhao et al.(2024)] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. WildChat: 1M ChatGPT Interaction Logs in the Wild. arXiv:2405.01470 [cs.CL] https://arxiv.org/abs/2405.01470