TrainVerify: Equivalence-Based Verification for Distributed LLM Training

Yunchi Lu (University of Michigan), Youshan Miao (Microsoft Research), Cheng Tan (Northeastern University), Peng Huang (University of Michigan), Yi Zhu, Xian Zhang, Fan Yang (Microsoft Research)

Enabling large models through scaling that are prone to silent errors

Llama 3.1
405 B

deepseek
671 B

Don't fit on one GPU

Model

Data

GPU 0

Single device

GPU 0 GPU 1 GPU 2 GPU 3

Tensor Parallelism (TP)

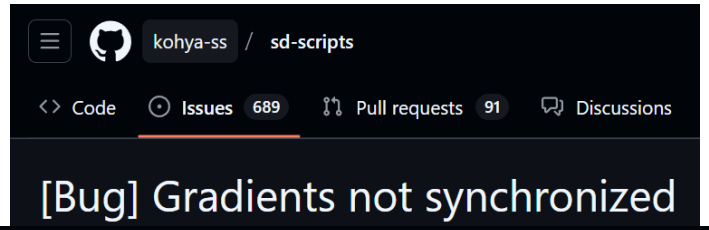
Scaling techniques are complex

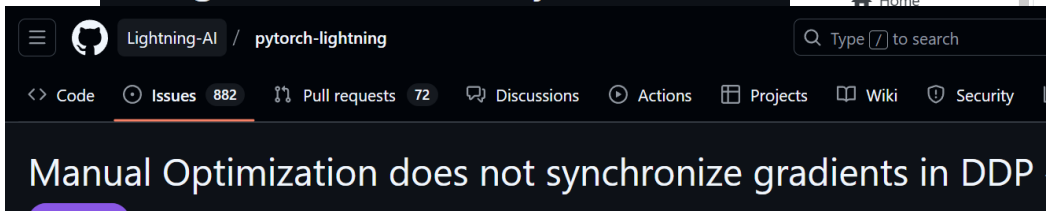
Sharding Communication

Optimizer Schedule

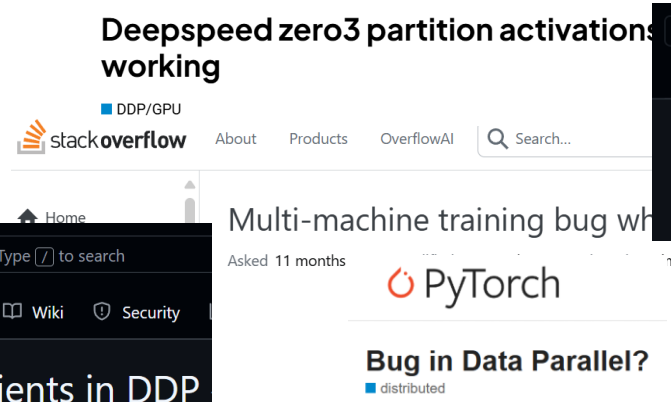
Prone to silent errors

Wrong communication operations

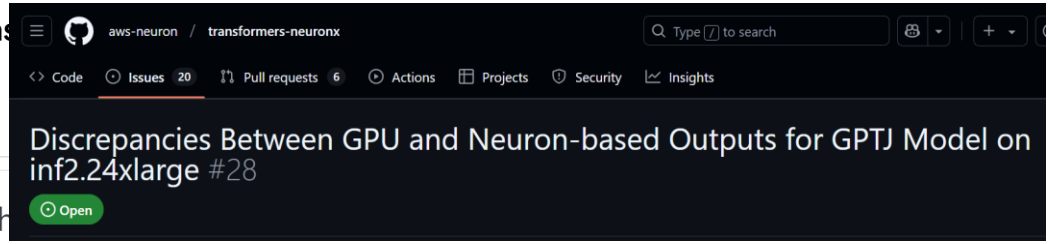




Wrong sharding



Wrong calculation



Behavior not like single-device pipeline, loss value not decreasing or garbage outputs

Model quality to drop

Example: Missing all-reduce (Megatron-LM)

LinearWithFrozenWeight backward fix when TP > 1, bug leads to non-decreasing loss

```
271  272          def backward(ctx, grad_output):
272  273              (weight,) = ctx.saved_tensors
273  274              grad_input = grad_output.matmul(weight)
274       -          return grad_input, None, None
     275  +
     276  +          if ctx.allreduce_dgrad:
     277  +              # All-reduce. Note: here async and sync are effectively the same.
     278  +              torch.distributed.all_reduce(grad_input, group=get_tensor_model_parallel_group())
     279  +
     280  +          return grad_input, None, None, None
```

Detecting when to do the all-reduce is difficult when manually looking through large ml-systems codebase

Silent bugs are tricky since they are subtle

Runtime recovery

CheckFreq [FAST ‘21], Varuna [EuroSys ‘22], GEMINI [SOSP ‘23], Oobleck [SOSP ‘23], Bamboo [NSDI ‘23], ReCycle [SOSP ‘24]

- ⊕ Fault-tolerant to failures
- ⊖ Relies on explicit error signals

Position: Expose silent errors before deployment

Testing frameworks

DeepXplore [SOSP ‘17], DeepTest [ICSE ‘18], Eagle [ICSE ‘22], NNSmith [ASPLOS ‘23], MLIRSmith [ASE ‘23], PolyJuice [OOPSLA ‘24]

- ⊕ Detects many bugs
- ⊖ No guarantee of absence of bugs

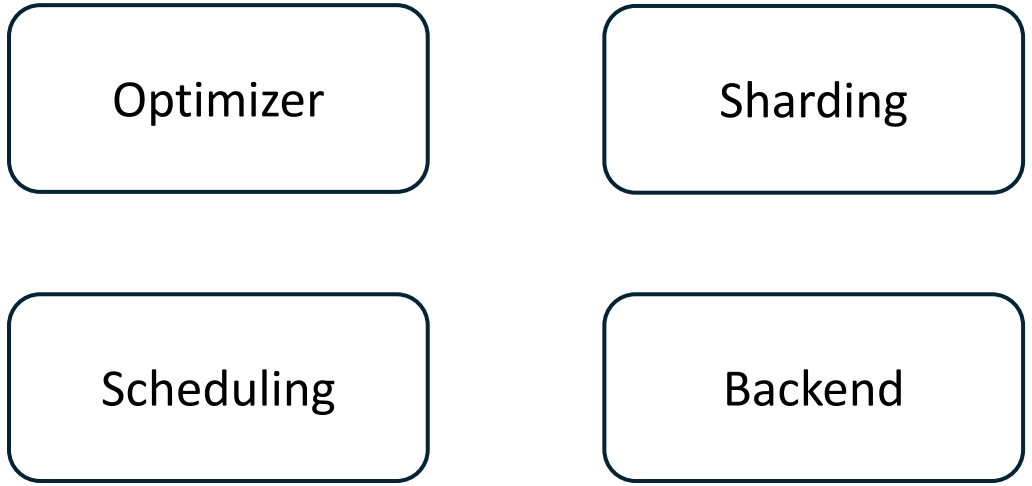
Position: Guarantee absence of errors in pipelines

Developers approach in debugging is ad-hoc

Examine intermediate tensor values in the entire huge code space manually

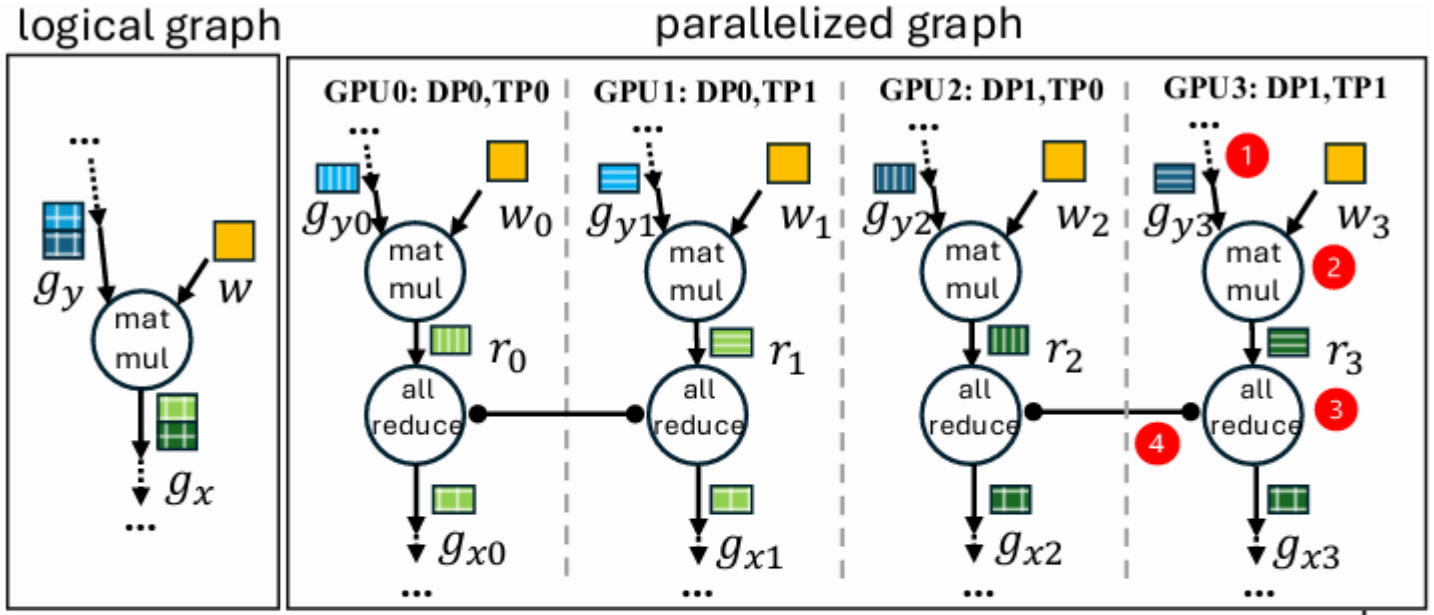
attention.py

```
print(...)
result = hlo.reshape(result, (n_seqs,
n_active_tokens, hidden_size))
print(...)
```



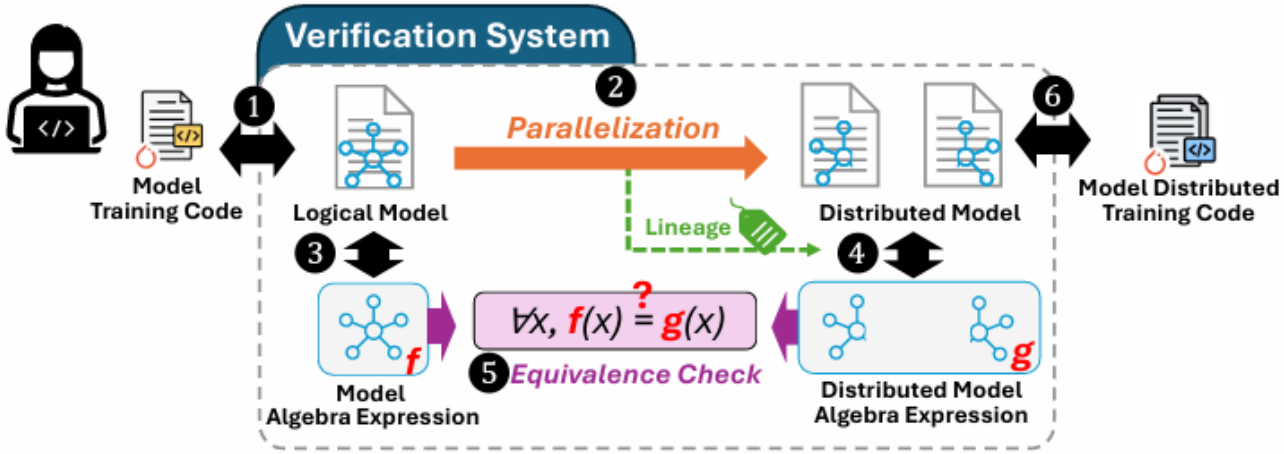
- Numerous amount of phases
- Hard to differentiate correct and wrong tensors due to floating-point round-off errors
- Tedious to manually piece tensors on multiple devices to match single on

Expose silent errors without explicit signals




```
def training_code_gpu3():
    model_tp_group = dist.new_group(ranks=[2,3])
 1  ... # upstream model code
    gy3 = ...
    w3  = ...
    gx3 = gy3.matmul(w3) 2
 3  dist.all_reduce(gx3,group=model_tp_group) 4
    ... # downstream model code
```

TrainVerify Workflow

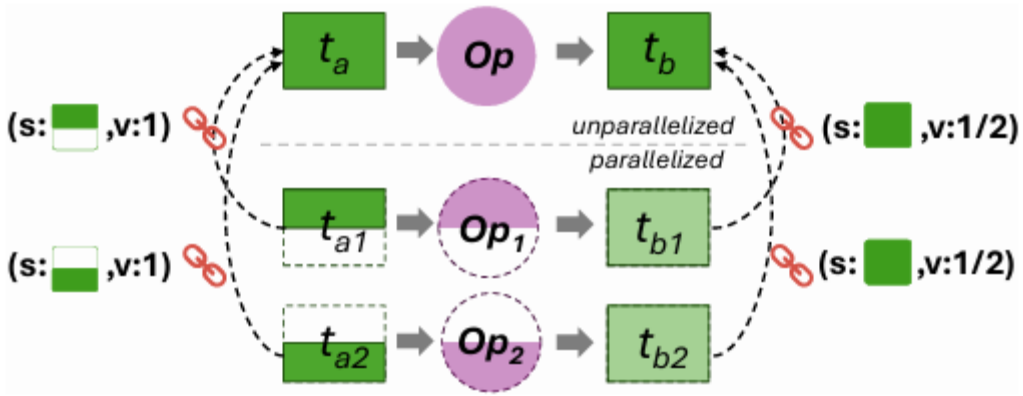


- Extract DFGs from logical and distributed (1) and (6)
- Symbolize DFGs from concrete tensors (3) (4) using lineage metadata (2) to track dependencies between logical and distributed
- Check equivalence using SMT solver (5)

Symbolic DFGs

- Construct manually DFG that represents the logical and distributed model which represent the forward and backward pass
- Define manually formal definition of operators in DNN frameworks
- Lineage: Tracks how tensors are related between single-device and distributed tensors

Lineage


t_a
Op
t_b
(s: ,v:1)
unparallelized
parallelized
(s: ,v:1/2)
t_{a1}
Op_1
t_{b1}
(s: ,v:1)
(s: ,v:1/2)
t_{a2}
Op_2
t_{b2}

Shape reduction

$$c_{1,1} = a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1} + a_{1,3} \cdot b_{3,1}$$
$$c_{2,2} = a_{2,1} \cdot b_{1,2} + a_{2,2} \cdot b_{2,2} + a_{2,3} \cdot b_{3,2}$$

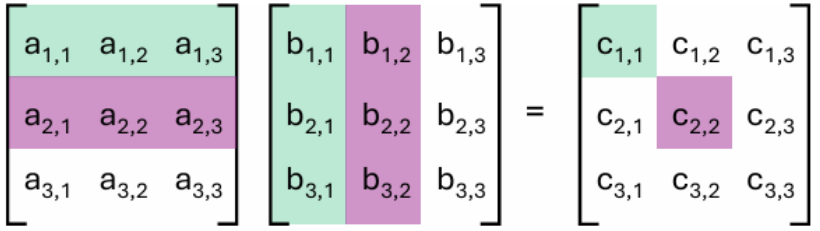


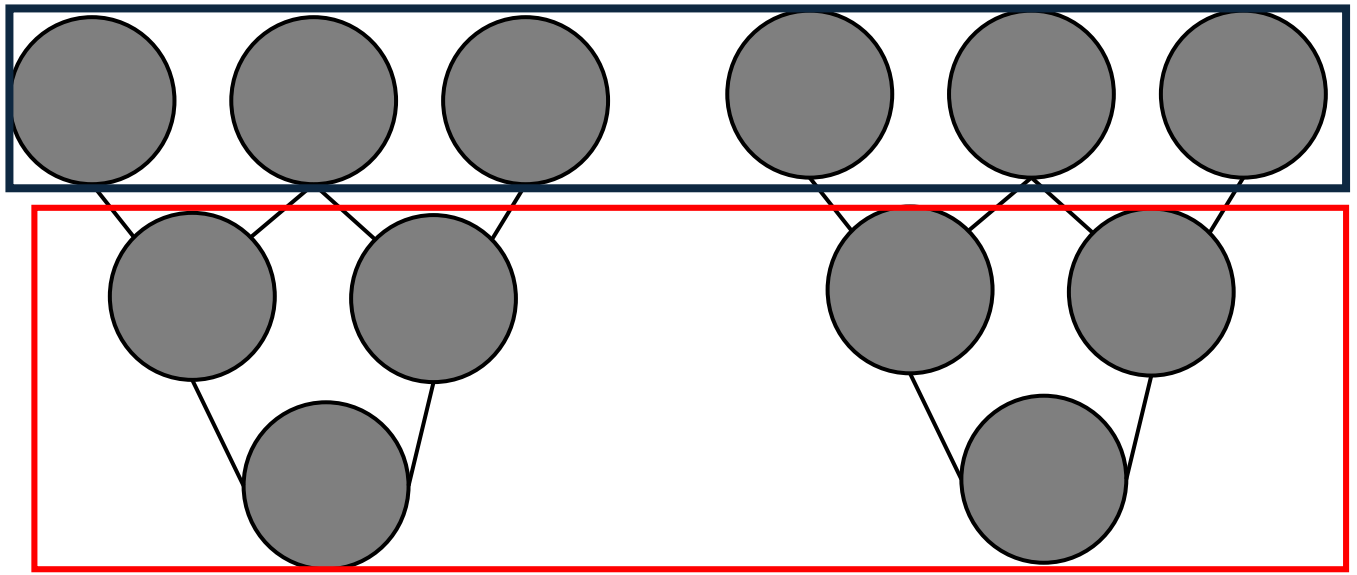
Figure 5. DNN operator `MatMul`: different output elements $c_{1,1}$ and $c_{2,2}$ are calculated using the same function but on different inputs.

- Operating on per-element values in tensor is expensive
- DNN operators have the SIMD (Single Instruction, Multiple Data) property, apply same computation across different data elements

Minimum shapes

- Shape alignment: reshape (M,N) to (M,P,Q) where N = PxQ
- Semantic intact: avoid reducing shape to 1 to avoid incorrectness in matmul

Stage-Parallel Verification



- Divide model into multiple stages and verify them in parallel
- Each stage consists of aligned subgraphs from single-device and distributed graphs, where each stage’s boundary is defined as tensors with lineage information

Implementation

- 6000 lines of code in Python
- Built on nnScaler, a distributed training framework from Microsoft
- Builds the single-device and distributed plans from generated IR graphs in nnScaler
- Tensor lineage is built using nnScaler’s indexing metadata and source code

Evaluation (Real-world models)

Exp. ID	Model	Layers	DP	TP	PP	NM
L1	Llama3-8B	32	512	1	1	1
L2	Llama3-70B	80	16	8	4	32
L3	Llama3-405B	126	64	8	16	16
D1	DS-V3-16B	27	16	4	2	16
D2	DS-V3-236B	60	16	8	4	16
D3	DS-V3-671B	61	32	8	8	16

Table 2. Evaluated real-world large models.

	L1	L2	L3	D1	D2	D3
Solver Parallelism	30	30	4	30	16	8
End-to-end Time	0.5h	7.5h	47h	0.5h	3.5h	31h

Table 3. Verification time for the evaluated models.

Evaluation (Setup)

- Dataflow graphs generated from machines with 4 A6000 GPUs
- Execution plans scaling up to 8192 GPUs

Evaluation (Scalability)

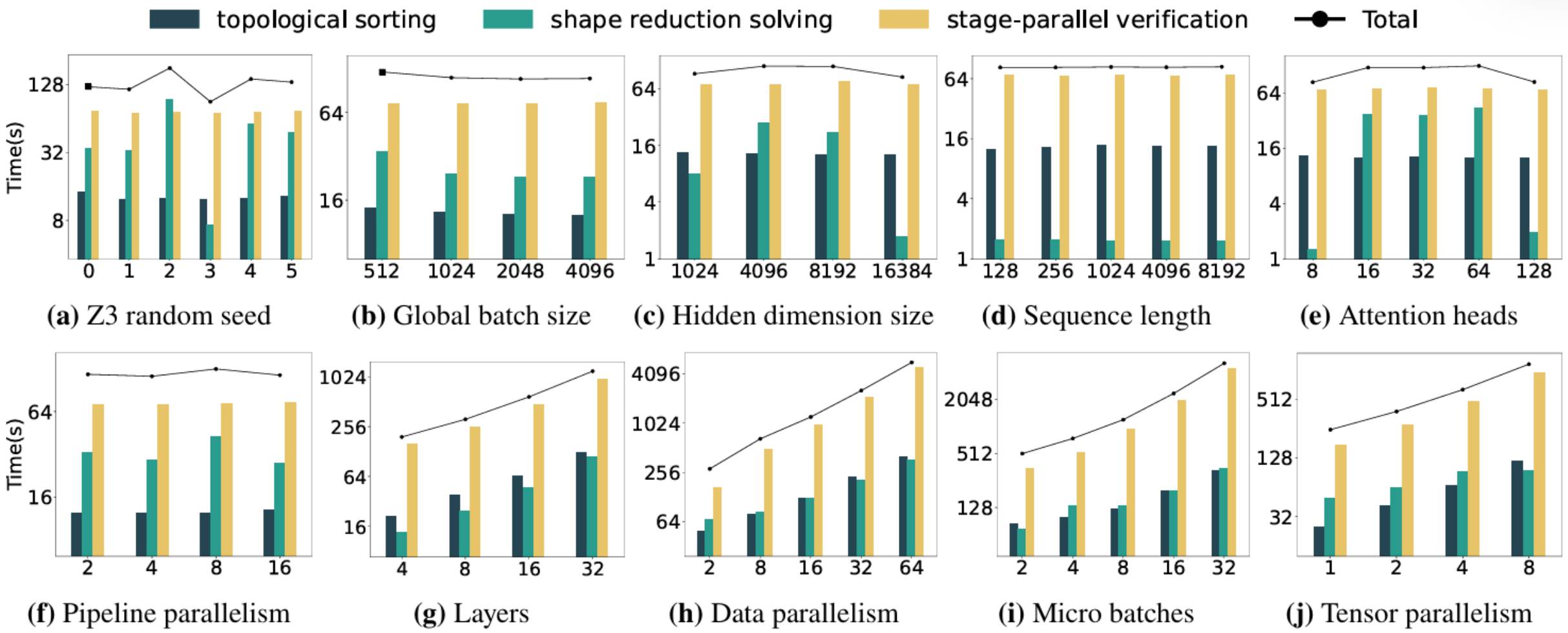


Figure 7. TrainVerify's performance trends regarding different training configurations. The y-axes use a $\log_2$ scale. Bars indicate the time breakdown by component, while lines represent the end-to-end verification time.

Evaluation (Scalability)

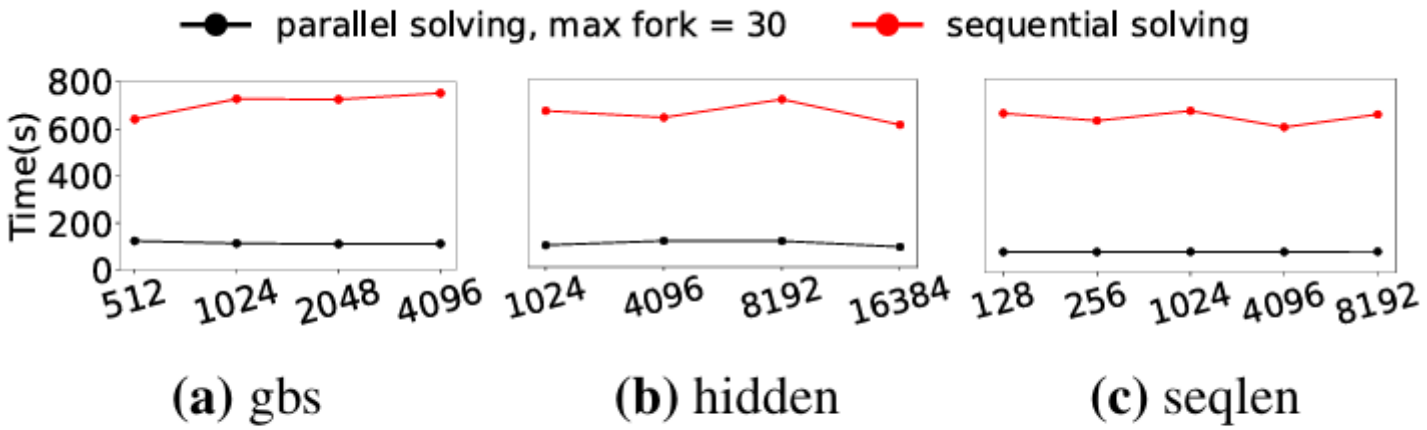


Figure 8. Verification time with vs. without stage parallelism.

Evaluation (Bugs eliminated)

- Incorrect communication operators
- Incorrect device assignment
- Incorrect partitioning
- Incorrect scaling
- Incorrect pipeline scheduling

- First four bugs are completely eliminated in TrainVerify, for the 5th bug, TrainVerify eliminates through early data dependency analysis

Evaluation (Bugs eliminated)

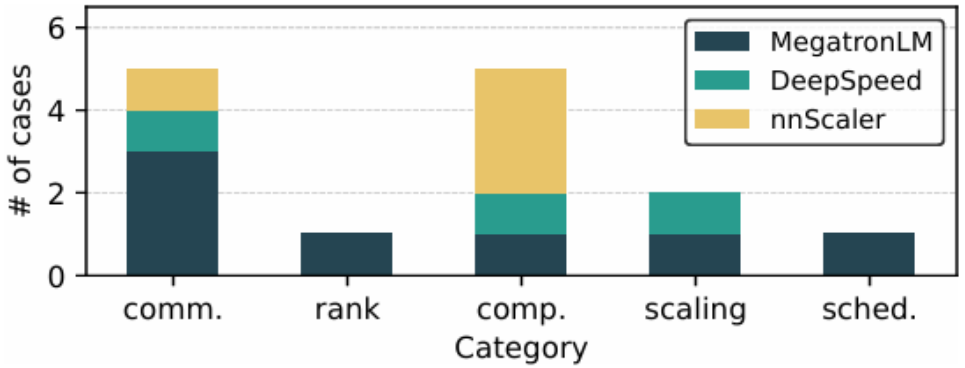


Figure 9. Reproduced incorrect parallelization cases.

- The bugs are reproduced in nnScaler, a machine learning training framework developed by Microsoft Research

Evaluation (New bugs)

- C1: Sharding a non-partitionable dimension
- C2: Dangling tensors in backward pass

Limitations

- Heavy reliance on graph-based execution plans
- Shape reduction assumes linearity on DNN operators
- Lineage information might be needed to be added manually

Conclusion

- Silent errors exist in many distributed training frameworks which are difficult to detect using existing work
- TrainVerify argues using the generated graphs, these errors can be exposed by verifying their equivalence
- TrainVerify scales to large models and exposes 14 old bugs and 2 new categories of bugs in machine learning training frameworks