A Comparison Between TorchFT and TorchPass for Fault Tolerant Training

TorchFT vs. TorchPass

TorchFT and TorchPass both address fault tolerance for distributed AI training, but in doing so, use fundamentally different mechanisms. TorchFT, developed by Meta, provides per-step fault tolerance, treating each training step as a distributed transaction and using a replica group as the failure domain. Upon failure, the entire replica group is removed and training continues without that replica group until it has been replaced.

TorchPass, by Clockwork, operates at the infrastructure layer and defines the failure domain down to an individual GPU. Upon failure, a live migration replaces only the affected resource, allowing training to continue at the same step with unchanged world size.

The following sections compare the two approaches along five important dimensions: the overhead imposed at each step, the impact failures have on training semantics, the failure domain cost, sparing requirements and the effort required to integrate with training code. See also the results of a benchmark comparison.

1. Per-step Overhead

TorchFT

Coordination Overhead

TorchFT uses a central Lighthouse for per-step coordination across replica groups and a per-group Manager for coordination within replica groups.

Each Manager sends periodic heartbeats to the Lighthouse which computes a quorum at every step. When it stops receiving heartbeats from a group, it marks that group as unhealthy and updates the quorum. This introduces a synchronization point at each step that wouldn't otherwise exist.

InIn order to reduce overhead, the coordination overlaps with backward pass computation. For training jobs where step times are measured in seconds, the overhead is typically negligible, but could become more significant for highly optimized training pipelines with sub-second step times.

replica group fails, TorchFT needs to tear down the cross-replica communication channel and rebuild it with the updated quorum membership. NCCL provides a ncclCommAbort function for this purpose, but ncclCommAbort can itself deadlock (see NCCL 2.29 documentation under fault tolerance) which makes it difficult to use with TorchFT.

Gloo Overhead

There is also a far more significant per-iteration overhead in TorchFT due to its use of the Gloo Collective Communication Library for cross-replica gradient all-reduce operations that happen on every training step in HSDP (Hybrid Sharded Data Parallelism) mode.

In conventional distributed training without TorchFT, the best practice for cross-replica all-reduce is to use NCCL (Nvidia Collective Communication Library) since it leverages GPUDirect RDMA, NVLink, InfiniBand kernel bypass, and pipelining across multiple NICs to achieve bus bandwidth of 370-400 GB/s in well-configured H100 and H200 clusters. In contrast, Gloo is a CPU-based library that operates over TCP/IP sockets and can typically sustain bandwidth of 10-40GB/s.

However, TorchFT must tolerate replica group failures that occur mid-step. When a

Gloo solves this limitation and can also be optimized for use with TorchFT by providing operation timeouts within 5 seconds when a peer fails, process group teardown and recreation in under 1 second with no risk of deadlock or corrupted state, and the surfacing of errors as standard Python exceptions that the training loop can catch and handle. The recommended TorchFT configuration is therefore hybrid: use NCCL inside each replica group and Gloo across replica groups (e.g. see Pytorch fault tolerance blog).

The downside of using Gloo is performance. Since Gloo operates over TCP/IP sockets, every gradient tensor must be copied from GPU memory to CPU memory, transmitted over TCP, and copied back to GPU memory on the receiving side at each iteration.

Published benchmarks quantify the gap. For example, Meta's PyTorch DDP paper (Li et al., 2020) reports that switching from NCCL to Gloo produced a 3x per-iteration slowdown for ResNet50 (~25M parameters) and a 6x slowdown for BERT (340M parameters), using 256 GPUs across 32 machines. Critically, the ratio widens as model and gradient sizes increase, because NCCL's bandwidth advantages compound at larger data volumes while Gloo hits the bandwidth ceiling of TCP and CPU memory.

The per-step overhead when using NCCL inside each replica group and Gloo across replica groups for a 13B model is 30-60%, as illustrated in the table below:

Metric	Medium Model (13B)
Gloo gradient all-reduce	5 – 15 seconds
NCCL gradient all-reduce	0.5 – 2 seconds
Base step compute time	~10 seconds
Gloo overhead as % of step	30-60%

TorchPass

TorchPass adds no per-step coordination overhead during training because TorchPass monitors the system asynchronously and intervenes only when needed for failures. Training runs at full speed. In the event a failure occurs, migration is triggered, after which training immediately resumes at full speed.

2. Training Semantics Under Failure

TorchFT

One of the big advantages of using TorchFT is that training doesn't pause when a failure occurs. Instead, when a failure occurs, the affected replica group is dropped from the quorum and training continues with fewer replica groups contributing to gradient averaging. However, this alters the training semantics since data samples assigned to the failed replica group are never processed. In particular, the effective batch size shrinks, some samples are dropped, the gradient statistics change, and the model sees a different distribution of training data than it would have without failures.

TorchFT treats this as acceptable because model convergence is generally robust to such perturbations. However, this makes fault-tolerant training with TorchFT fundamentally not equivalent to non-fault-tolerant training as the model trained under failures will have seen different data and experienced different gradient dynamics than one trained without failures. The extent of these differences will be amplified if the failure is not remediated quickly.

TorchPass

TorchPass aims to resume training from exactly where it left off. The training trajectory after failure will be identical to what it would have been without the failure. No data samples are skipped, no gradient contributions are discarded, and the model sees exactly the training distribution it was designed to see.

3. Parallelism

TorchFT

TorchFT achieves per-step fault tolerance by using replica groups as failure domains. Each replica group runs a standard distributed training stack and TorchFT uses a fault-tolerant DDP (distributed data parallel) protocol across replica groups to synchronize updates and recover failed groups by transferring state from healthy replicas. TorchFT has out-of-the-box support for DDP and HSDP (Hybrid-Sharded Data Parallelism and experimental support for DiLoCo (Distributed Low-Communication) and LocalSGD (Local Stochastic Gradient Descent).

When configuring distributed training jobs, the model developer must select a parallelism configuration based on fault tolerance requirements (e.g. failure domain size) rather than one designed for optimal performance and resource utilization. In practice, this can have significant performance trade-offs, especially in smaller-sized clusters.

TorchPass

When using TorchPass, parallelism can be configured to optimize for performance and utilization rather than to optimize for fault tolerance.

4. Failure Domain Cost

TorchFT

As described above, TorchFT uses an entire replica group as the failure domain (when a failure occurs the impacted replica group is removed from the training).

Consider a training job using 256 GPUs configured as four replica groups, each with 64 GPUs. When a single GPU fails, the cost of the failure is 64 GPUs. The 63 healthy GPUs in that replica group sit idle while the replica group remains in a failed state.

TorchPass

TorchPass uses exactly the failed resource as the failure domain.

Using the same example as above (256 GPUs, four replica groups, 64 GPUs per replica group), when a single GPU fails, its state is migrated to a healthy GPU which joins the replica group and training continues at the same step. In other words, the remaining 63 healthy GPUs continue contributing along with the replacement GPU.

5. Spares

TorchFT

By design, TorchFT does not require spare resources. When a failure occurs, the impacted replica group is immediately removed from training and samples assigned to that replica group will not be processed. If spare resources for an entire replica group become available at any time, the replica group will be added back into the training job.

TorchPass

TorchPass requires spare resources to perform live GPU migration and keep training running after a failure. Because the failure domain is limited to just the affected resource (down to a single GPU) spare capacity only needs to match the size of the failure.

While most production environments already maintain standby nodes, TorchPass does not require dedicated spares as it can dynamically draw replacement capacity from any available resources or even reclaim it from lower-priority jobs.

6. Integration

TorchFT

TorchFT requires the model developer to make the following changes:

- A Manager object must be instantiated to coordinate with the Lighthouse.
- The standard DDP wrapper must be replaced with TorchFT's DistributedDataParallel
- The optimizer must be wrapped with TorchFT's Optimizer class.
- The developer must provide state serialization callbacks for peer-to-peer weight recovery
- The training loop must be restructured to account for the fact that step counts can regress on rollback.

Note on using TorchFT with training frameworks: TorchFT currently supports TorchTitan through a deep integration that makes use of micro-batching. Although TorchFT provides a DDP wrapper for use with integrations, the wrapper won't work for micro-batching which means that supporting other frameworks like Megatron-LM or Deepseek with micro-batching would require a significant effort, including a deep system-level understanding of the respective frameworks.

TorchPass

TorchPass requires the model developer to make the following changes:

- Import the TorchPass library
- Register their existing checkpoint save and load functions
- A one line change to wrap the body of each training step with a TorchPass context manager

TorchPass is designed to support popular frameworks including TorchTitan, Megatron-LM and DeepSpeed, without requiring an in-depth understanding of those frameworks.

Conclusion

TorchFT and TorchPass represent two effective approaches for achieving training reliability. TorchFT is optimized to keep training running through failures and accepts the per-iteration overhead, altered training semantics and coarse-grained failure domains as acceptable costs. TorchPass prioritizes semantic preservation, surgical resource replacement and zero per-iteration overhead by accepting that migrations will briefly pause training.

For more information, see: Fault Tolerance Benchmark: Clockwork TorchPass, TorchFT and checkpoint restart

Industry and Customer Voices

"In our testing, Clockwork.io TorchPass delivered the fastest and most efficient fault-tolerant performance for a gpt-oss-120B training run. We used TorchTitan on a Kubernetes cluster with 64x H200 GPUs. During our testing we measured job completion time (JCT) and Model FLOPs Utilization (MFU) against a standard approach (checkpoint-restart) and the leading open-source fault-tolerant training framework (TorchFT). We simulated multiple hardware failures on the cluster in order to stress test the fault-tolerant training frameworks.

When compared to checkpoint-restart, TorchPass was significantly faster to recover from failures. This reduced overall JCT and maintained high MFU. And when compared to TorchFT, TorchPass had a significantly higher MFU. This reduced overall JCT while also maintaining an equal time to recover from failures.

Using TorchPass also has a downstream effect where it provides users with an opportunity to reduce or even remove checkpointing from their training code. This means larger effective batch sizes, lower risk of out of memory errors (OOMs), and less time spent thinking about storage. For a research organization, this can ultimately mean a faster time to reach their training objective," concluded Nanos.

Jordan Nanos, Member of Technical Staff and lead author of ClusterMAX—SemiAnalysis

---

"As Blackwell clusters roll out with an NVL72 domain, and we look to the future with Rubin Ultra's NVL576 domain, the idea that a single GPU error or network link flap can take down an entire run is totally unacceptable," said Patel. "TorchPass solves a huge challenge with cluster reliability: it provides transparent failover and live workload migration that keeps MFU high, which in turn drives better GPU economics."

Dylan Patel, Founder and CEO, SemiAnalysis

"Managing compute output across large-scale GPU clusters is vital to ensuring we're delivering reliable capacity to our customers. By using TorchPass we have the support of a company that focuses on resilience like it is a core business function: it replaces any specific failing GPU and keeps the rest of the job moving, rather than making one small problem impact our large-scale operations. In our evaluation, Live GPU Migration preserved both run continuity and throughput under real fault conditions, which is exactly what you need to deliver predictable time-to-train and a better customer experience at scale."

David Power, CTO of Nscale