LSAI2025 Feature-Report: Distributed Checkpointing and Timeaware exit with DDP

Raphael Kreft
rkreft@student.ethz.ch

Shaswat Gupta
shagupta@student.ethz.ch

Abstract

Checkpointing is a vital component of large-scale AI training, as with increasing runtime or number of machines, the frequency of hardware failures or preemptions also rises. We implement two checkpoint techniques (vanilla and distributed) and time-aware checkpoint logic. We test the implementation in a Distributed Data Parallel setting and show that distributed checkpointing is more time-efficient. Furthermore, checkpointing in general does not affect the weights upon load and store and the loss-convergence is equal. The code is available at https://github.com/Shaswat-G/PyRecover

1. Introduction

The training of modern artificial intelligence models, particularly large-scale foundation models, demands immense computational resources. AI research laboratories routinely employ thousands of GPUs to train transformer-based architectures over weeks or even months. While this level of scale is essential for state-of-the-art performance, it introduces significant engineering challenges—chief among them, the reliability and fault tolerance of long-running training jobs.

As the number of machines involved in training increases, so does the likelihood of hardware failures, preemptions, or system interruptions. These failures can result in substantial loss of progress. Checkpointing, which involves periodically saving the state of the training process, is therefore a necessity. It ensures that, in the event of a disruption, training can resume from the last saved state with minimal loss of compute and time.

In the context of the Large Scale AI Engineering project, we explored this critical aspect by implementing and evaluating multiple checkpointing strategies within a distributed training setup using PyTorch. Our experimental framework involves a decoder-only transformer model trained with a Data Distributed Parallel (DDP) configuration.

Our key contributions in this work are threefold:

1. **Vanilla PyTorch Checkpointing**: saves the model weights, optimizer state, learning rate scheduler state, data loader state, and metadata such as the current training step and epoch—all serialized using standard PyTorch utilities.
2. **Distributed Checkpointing**: efficient checkpointing using *torch.distributed.checkpoint* module. Significantly reduces I/O bottlenecks and hence speeds-up checkpointing.
3. **Time-Aware Checkpointing**: monitors the remaining wall time allocated for a training job and triggers a final checkpoint before the job is terminated. This ensures that progress made during the last phase of training is not lost due to job timeouts.

We implement and benchmark these checkpointing techniques, analyzing their impact on training runtime and resource usage. Our experiments demonstrate that both distributed and time-aware checkpointing strategies can be integrated without affecting the model weights upon load or store or the loss convergence. The effect on training run-time, of course, depends on the chosen checkpoint frequency. These results highlight the practicality and necessity of checkpointing in large-scale AI model training.

2. Method

This section details the implementation of our checkpointing strategies, as developed in the context of a distributed training pipeline for transformer models using PyTorch.

The training framework is designed for flexible experimentation through command-line arguments and is integrated with SLURM job scheduling. While single-GPU runs are supported, the system explicitly supports multi-GPU and multinode training and checkpointing using a `DistributedDataParallel` (DDP) implementation/setup. More information about options and usage of our implementation can be found in the ReadMe of the linked GitHub-Repository.

2.1. Vanilla Checkpointing

We implemented a standard checkpointing method that makes use of PyTorch default *torch.load* and *torch.store*

methods. We save model weights, optimizer state, learning rate scheduler state, data loader state (needed because of distributed sampling), and current training step and epoch. This ensures full reproducibility and continuation after interruption.

Storing The rank 0 process saves the full state in a single file (approximately 45 GB for the default model) within a configurable experiment folder.

Checkpoint Verification We optionally compute an MD5 hash of the entire checkpoint file post-creation and store it in a separate file for each checkpoint. This hash can be verified upon loading to detect file corruption. This feature makes sure that model checkpoints are not corrupted and training can resume and converge as expected without confusing dow-stream errors.

Loading We evaluated two strategies during development. In the first approach the proccess being rank 0 loads and verifies the full checkpoint and then broadcasts the full checkpoint from rank 0 to all other processes. This approach was discarded due to the long checkpointing times ¿6min in initial experiments. This is likely due to the communication of the large checkpoint across processes as indicated by NCCL timeouts. The second and chosen approach supports this finding. It allows each rank to independently load the checkpoint file. This worked by a factor of 3 faster. We implement multiple improvements to make checkpointing even faster: To avoid I/O congestion, the ranks stagger their reads using predefined per-rak wait times. If verification is active, rank 0 performs asynchronous checksum verification in a separate thread. Using these optimizations, the vanilla checkpointing has a reasonable time efficiency.

2.2. Distributed Checkpointing

Using `torch.distributed.checkpoint`, we implemented another checkpointing approach that writes the same training state as the vanilla approach. This method shards the full checkpoint state across multiple files and thus distributes the I/O load across ranks and prevents laod congestion on one big file. Each process saves and loads a subset of the full state, synchronizing with others via distibuted communication primitives to reconstruct the full state. This significantly reduces I/O bottlenecks and improves overall efficiency. Currently, this method does not support checkpoint verification. We also note that this way of checkpointing integrates more easily with other distributed training modi in PyTorch such as FSDP.

2.3. Time-Aware Checkpointing

Time-aware checkpointing is designed to minimize wasted computation (meaning updates to model-weights that are not checkpointed before a job ends) by ensuring a final checkpoint is saved just before a SLURM job's wall-time expires. This feature dynamically adapts to the observed performance of the training run.

The system tracks 1) the maximum observed iteration time and 2) the maximum observed checkpoint save time to dynamically compute the buffer time. This buffer-time then acts as a safe margin for the final checkpoint invocation. The buffer time throughout training is computed as:

$$\text{buffer\_time} = k_1 \cdot \text{max\_iter\_time} + k_2 \cdot \text{max\_ckpt\_time} \quad (1)$$

where k_1 and k_2 are empirically chosen safety multipliers (e.g., $k_1 = 10$, $k_2 = 2$ in our implementation). This adaptive approach ensures robustness to fluctuations in system and I/O performance, always reserving enough time for a final checkpoint.

The SLURM job end time is read from the `SLURM_JOB_END_TIME` environment variable, set by the submission script. At the start of each training step, rank 0 checks if for the remaining time t: $t$max_iter_time + max_ckpt_time + buffer_time. If so, a `should_stop` flag is set and broadcast to all ranks. A final checkpoint is stored and all processes exit gracefully. This ensures no progress is lost due to job termination. When the flag is off, the rest of the training and checkpointing logic is untouched, so there is no performance penalty.

3. Experiments

A suite of eleven experiments was devised to quantify the cost-benefit trade-offs of the checkpointing strategies we present in Section 2. Furthermore we proof that the model weights are not affected by checkpointing and the loss convergence is similar. For this we run eleven training runs with different configurations and keep track of runtime, checkpointing time, iteration time as well as the loss values and we comapre the final checkpoints to enusre weight equality (for non-time-awasre runs).

Unless otherwise stated, each training job executed **3,000 iterations** under a nominal **40-min wall-time budget**. We run for many iterations to effectively show the effect of checkpointing. Hence we must restrain our setup to 4 GPUs maximum, as otherwise the job time limit as set for the project is insufficient.

For time-aware runs we set the checkpoint interval to 1,000 iterations but vary the SLURM allocation with 40min, 30 min and 20 min, respectively, thereby forcing an early termination and exercising the proposed time-aware checkpointing. Table 1 shows the configurations of our 11 experiment runs.

Table 2 shows the results of all runs. The job name entails

Table 1. Experiment groups and variables

Group	Runs	Configuration	Variables
Baseline	1–2	No checkpointing (single-GPU vs. 4-GPU DDP)	–
Frequency sweep	3–8	Vanilla vs. distributed checkpointing	Checkpoint intervals $\in \{250, 500, 1\,000\}$ iterations
Time-aware	9–11	Distributed checkpointing (interval = 1,000)	Wall-time budgets $\in \{40, 30, 20\}$ min

Table 2. Raw experimental results

job_name	#nodes	#tasks	steps	wall-clock [s]	iter time [s]	ckpt freq	#ckpts	ckpt time overall [s]	ckpt time avg [s]
Single-GPU baseline	1	1	3,000	999.0	0.33	–	0	0	–
Single-GPU, vanilla, 250	1	1	3,000	1,420.4	0.33	250	12	428.6	35.7
Single-GPU, vanilla, 500	1	1	3,000	1,201.6	0.33	500	6	209.9	35.0
Single-GPU, vanilla, 1000	1	1	3,000	1,089.5	0.33	1,000	3	105.0	35.0
4-GPU DDP baseline	1	4	3,000	1,202.6	0.40	–	0	0	–
4-GPU DDP, dist, 250	1	4	3,000	1,325.0	0.40	250	12	119.1	9.9
4-GPU DDP, dist, 500	1	4	3,000	1,266.5	0.40	500	6	61.4	10.2
4-GPU DDP, dist, 1000	1	4	3,000	1,247.5	0.41	1,000	3	30.6	10.2
4-GPU DDP, timeaware, 20m	1	4	2,720	1,128.8	0.40	1,000	3	33.1	11.0
4-GPU DDP, timeaware, 30m	1	4	4,240	1,728.4	0.40	1,000	5	52.4	10.5
4-GPU DDP, timeaware, 40m	1	4	5,000	2,020.3	0.39	1,000	5	51.2	10.3

the setting: first the number of GPUs (equal to the column #tasks), then the type of checkpointing used:

- baseline: no checkpointing used
- vanilla: vanilla checkpointing method is used
- dist: distributed checkpointing is used
- timeaware: time aware checkpointing is activated using distributed checkpointing.

This is followed by either the checkpointing frequency given as the number of training step iterations or, in the case of time-aware runs, the set job duration. The iteration time is given as average in seconds.

We log the loss of each training run and compare the convergence of the loss value across experimental runs. This is depicted in figure 1.

4. Discussion

This section reviews the eleven experimental runs in Table 2, separating *pure compute time* (average iteration time $\times$ number of steps) from *checkpoint overhead* (wall-clock minus compute) to quantify the cost of fault tolerance during large-scale Transformer pre-training on an HPC cluster.

4.1. Baseline runs

With identical global batch size, Distributed Data Parallel (DDP) increases the per-step latency from 0.33 s (single GPU) to 0.40 s, a 20 % penalty attributable to gradient synchronisation and host–device transfers. At the same time the amount of processed data increases linearly with the numebr of GPU's used. These two baselines anchor all subsequent comparisons.

4.2. Single-GPU checkpointing

Vanilla PyTorch checkpoints are fixed at 45 GB. Because checkpoint time is dominated by I/O, runtime overhead scales linearly with the number of checkpoints. At a cadence of 1 000 steps, wall-clock time rises by 9 %, giving a recovery granularity of roughly 30 min (1 000 $\times$ 0.33 s). The measured 35 s per checkpoint agrees with the theoretical SSD throughput of 45 GB $\div$ 1.3 GB s^{-1}.

4.3. Sharded distributed checkpointing

Using `torch.distributed.checkpoint` reduces checkpoint latency from 35 s to about 10 s— a 3.5$\times$ improvement—and cuts total overhead from 42 % to between 2.5 % and 9 %, depending on frequency. Even at an aggressive 250-step cadence, wall-clock time grows by only 10 % relative to the DDP baseline. Per-checkpoint variance stays below 3 %, indicating well-balanced I/O.

4.4. Time-aware checkpointing

When wall-time limits are shortened, the controller triggers a final checkpoint whose overhead remains below 11 s. For a 20-min allocation, the mechanism saves 720 steps (26 % of the run) at a cost of 33 s—about a 9$\times$ return on time invested. With a 30-min limit, 240 steps are preserved for an additional 52 s. Under a 40-min limit the job completes normally, and the safeguard is invoked only once, confirming the effectiveness of the back-off logic.

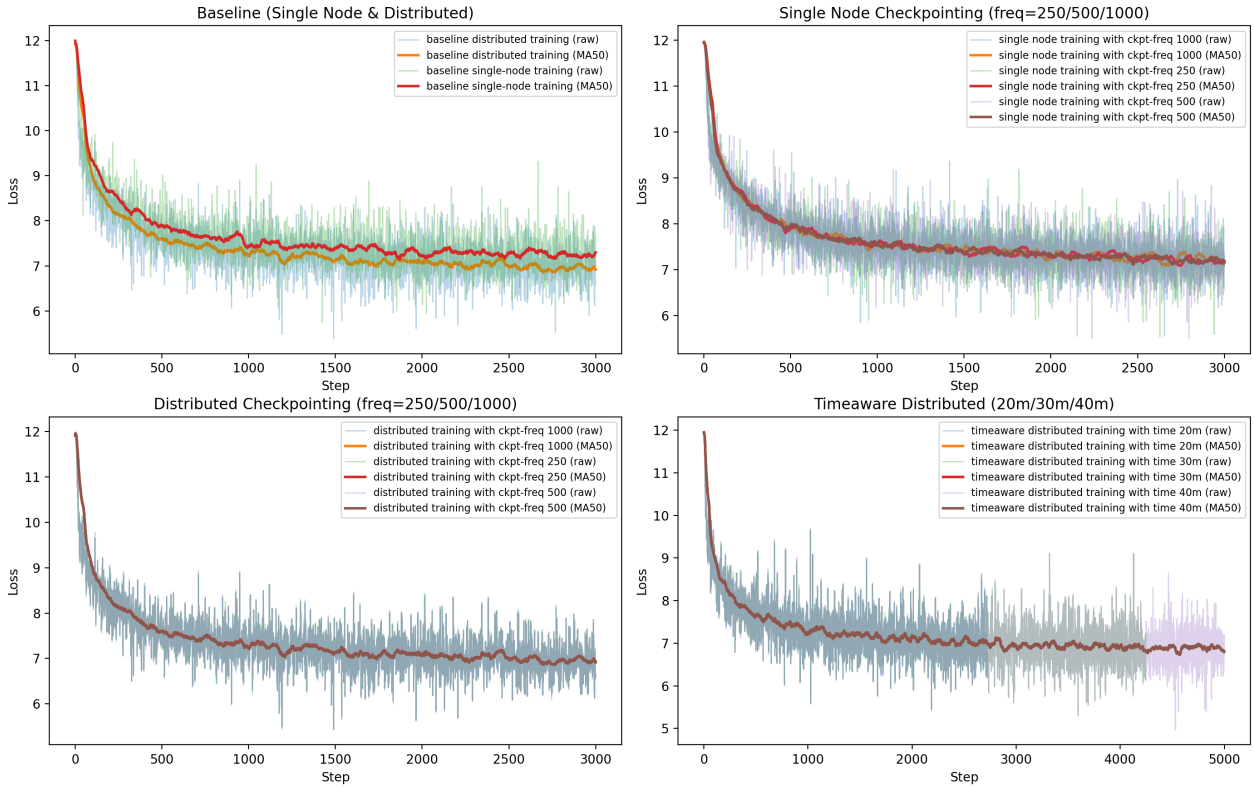

Figure 1. Loss vs. training steps for all 11 experimental runs. Each subplot shows the loss trajectory for a specific configuration, with both raw and moving-average smoothed curves.

4.5. Key observations

Checkpoint latency, rather than frequency, dominates overhead; sharding(distributed checkpointing) reduces latency from 35 s to 10 s, thereby lowering wall-clock penalties by a factor of 3–4. Here we note that the difference is even more exaggerated when having multiple nodes in the training as some of our early experiment showed. A 1 000-step cadence establishes a Pareto frontier: $\leq$10 % slowdown for vanilla checkpointing implementation, $\leq$3 % for the distributed checkpointing, while still providing 11–15 min recovery granularity. Time-aware termination amortizes its own computational cost, delivering $>6\times$ net savings whenever a job faces pre-emption.

Compute performance scales sub-linearly with GPU count ($0.33 \rightarrow 0.40$ s), whereas checkpoint overhead grows super-linearly unless properly sharded; consequently, scalability claims must account for both compute and I/O characteristics. MD5 verification in the vanilla path requires < 0.5 s per checkpoint and is notably absent from the distributed API; asynchronous checksum computation could bridge this gap with negligible additional overhead. The low overhead in the vanilla implementation is achieved through the asynchronus check in a separate thread.

5. Future Work

The present study exposes several avenues for advancing both distributed and time-aware checkpointing across four key themes. *Policy-aware job resubmission* could integrate automatic pipelines that inspect final checkpoints, verify MD5 hashes, and submit follow-on SLURM jobs, while runtime prediction using observed throughput would enable better resource estimation and code-base drift detection would prevent inconsistent runs. This auto-resubmission could be implemented with deamons or slurm job chaining whereas code drift could work with git commit hashes. *Adaptive checkpoint scheduling* would maintain rolling estimates of compute throughput and I/O latency, adjusting intervals to keep projected overhead below user-specified budgets through feedback-driven mechanisms and error-aware back-off strategies that implement exponential retry patterns. This would improve configurability and resilience.

I/O and storage innovations could use asynchronous snapshots that overlap gradient computation with background streaming, differential, and compressed checkpoints storing only parameter deltas, and hierarchical storage policies exploiting burst buffers with automated decisions between NVMe, Lustre, or S3 based on bandwidth availability.

Broader robustness and usabilityof the ladder would include failure prediction models applied to historical node telemetry for proactive checkpoint triggering, heterogeneous restart capabilities that handle changes in world size or GPU type between jobs, and improved observability through Prometheus and Grafana dashboards that provide real-time monitoring. Realizing these extensions would transform the system from a fault-tolerant prototype to a fully autonomous training service capable of executing multi-week runs with minimal operator intervention.