



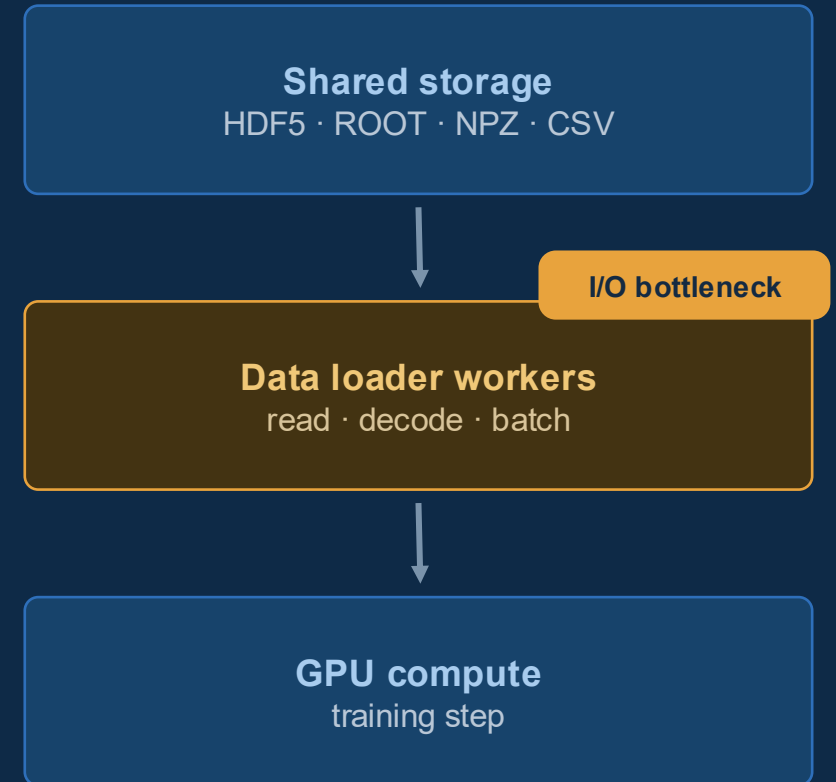
Characterizing I/O Performance for ML Data Loaders at Scale Using Darshan

Bradley Leesman

C2P2 Annual Seminar

Why Data Loading Matters in HEP ML

- Modern HEP ML workloads rely on large-scale, detector-level datasets
 - individual event sizes are substantial — and overall data volume grows rapidly
- Data loading has emerged as a critical bottleneck
 - often one of the most I/O-intensive stages of the training pipeline
- Loading speed directly constrains training at scale:
 - achievable batch size and data ingestion rates
 - parallelism and training throughput on HPC systems
- GPUs are only as fast as the data pipeline that feeds them



loading throughput gates how fast the GPUs can be fed

Project Goals

1

Benchmark ML data loaders at scale

The native PyTorch DataLoader and a ROOT-based data loader for PyTorch (native ROOT / RNTuple support), exercised under realistic ML training conditions on Perlmutter.

2

Characterize I/O behavior across HEP data formats

ROOT RNTuple, HDF5, NumPy NPZ, and CSV — with large-scale I/O traces collected via Darshan, a lightweight and scalable I/O profiling framework.

3

Derive actionable optimization insights

Identify performance bottlenecks and characterize access patterns to optimize data-loading pipelines for large-scale ML training — and, as an extension, learn to predict them.

Our Instrument: The Darshan I/O Profiler

- A lightweight, scalable I/O profiling framework for HPC
 - records the file activity of every process with negligible overhead
- Transparent — no modifications to the ML code under study
 - attaches when a process launches and observes its POSIX, STDIO, and HDF5 activity
 - the measurement itself does not distort the behavior being measured
- Yields a complete I/O profile per process
 - operation counts, bytes moved, access sizes, and timings, file by file
- Lets us ask precisely where data-loading time goes at scale



training code runs unmodified

Study Design: Loaders, Formats, and Questions

Data loaders under comparison

PyTorch DataLoader (native)

The ML-community default — format-agnostic, feeds the GPU through parallel worker processes.

ROOT-based loader for PyTorch

A recent development, HEP-native — reads ROOT and RNTuple structures directly, no conversion step.

Data formats under comparison

ROOT RNTuple ROOT's next-generation columnar format

HDF5 chunked hierarchical binary — today's FTAG training format

NumPy NPZ zipped arrays, common in ML prototyping

CSV row-oriented text — the simplest baseline

Measured from the traces: throughput · latency · I/O contention · access patterns

Which loader × format combinations keep GPUs fed at training scale? Where does loading time actually go?

Controlled Benchmark: Isolating Loader Parallelism

- A deliberately small PyTorch benchmark: FashionMNIST + a simple MLP
 - with modest compute, data loading dominates — I/O behavior has nowhere to hide
- One knob at a time: sweep DataLoader worker parallelism
 - batch size and epochs held fixed; every configuration traced with Darshan
 - the exact grid of worker counts is still open — it will adapt as early traces come in
- The questions this testbed answers
 - how does loader parallelism reshape the I/O stream — and where does it stop helping?
 - a clean baseline to interpret the production workload against

Worker-parallelism sweep — initial grid

2

4

10

12

14

*candidate values from the current test harness —
not locked in; the final grid follows the early results*

*one Darshan trace per configuration —
same data, same model, only parallelism varies*

The salt Framework: How ATLAS Trains Its Taggers

salt — the ATLAS flavour-tagging (FTAG) group's training framework

config-driven: one YAML defines model, data, and trainer (`salt fit --config GN2.yaml`)

- Built on PyTorch Lightning

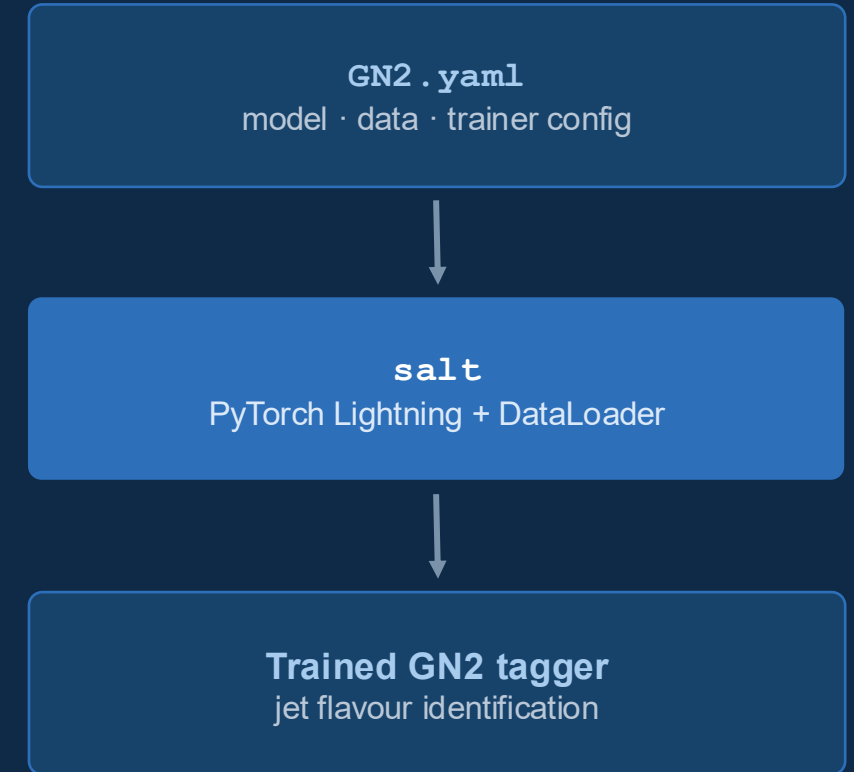
- multi-GPU, multi-node distributed training comes out of the box
- batches flow through the standard PyTorch DataLoader — exactly the I/O path this project studies

- Reads the FTAG group's preprocessed HDF5 samples

- per-jet arrays sharded across split files, plus normalization and class dictionaries

- Here it trains GN2 — ATLAS's current-generation jet flavour tagger

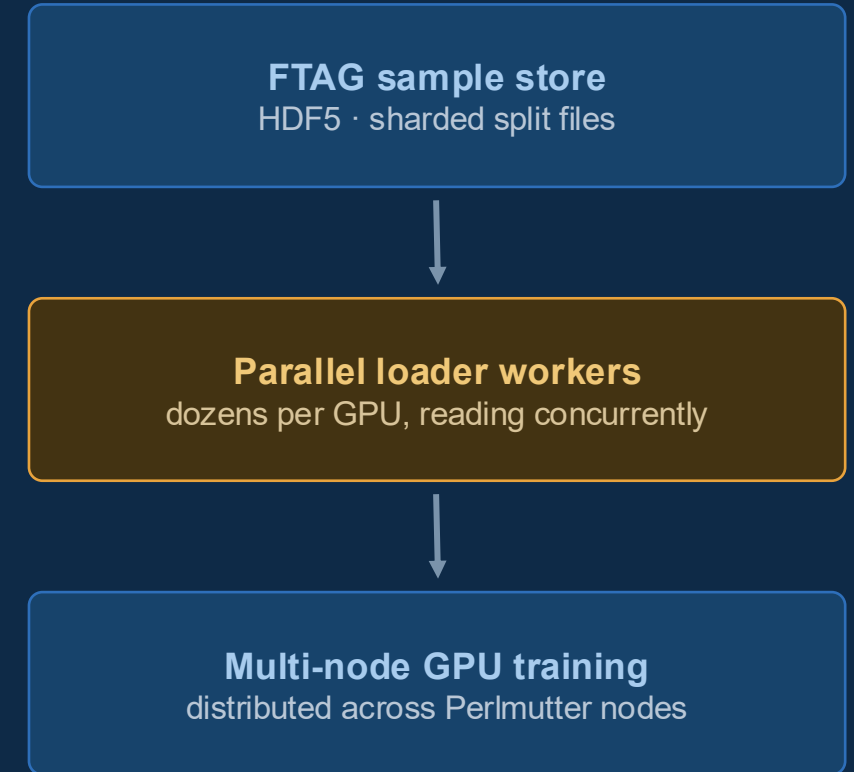
- so our traces reflect a real ATLAS training campaign, not a toy



*instrumenting salt = instrumenting the real
ATLAS training I/O path*

Production Workload: ATLAS GN2 Training

- A real HEP training campaign as the second testbed
 - GN2 trained with salt on the FTAG HDF5 samples — a 260 M-sample training set
- Realism the small benchmark cannot provide
 - many loader processes reading concurrently from shared storage
 - the regime where contention and access-pattern effects actually appear
- Scaling up in stages
 - reduced slices first, to validate the instrumentation
 - the final production scale is still being decided — the traces will inform it
- Together, the two testbeds bracket the space: controlled isolation ↔ production reality



current working config: 4 nodes × 4 GPUs, 32 workers per rank — final scale TBD

From Traces to Answers: What We Look For

Access patterns

Large sequential streams, or storms of small random reads?
Operation sizes and offsets show which regime each loader ×
format lands in.

Read amplification

How many bytes hit storage per byte the model consumes?
Chunking, compression, and layout can multiply the raw I/O.

Parallel efficiency

Do 2× the workers deliver 2× the ingest? Per-process profiles
expose stragglers and serialization among loader workers.

Contention at scale

What happens when many processes share one filesystem?
Timing overlap across ranks reveals interference no single-
process test can show.

Each question is directly computable from Darshan's per-process operation counts, access sizes, and timings.

Current Status



Darshan verified on Perlmutter

loads cleanly into PyTorch training processes — training code unmodified



Tracing wired into both testbeds

every training process and loader worker produces an I/O profile



First reduced-scale GN2 training runs

2 M-sample slices exercising the full distributed setup (16 GPUs)

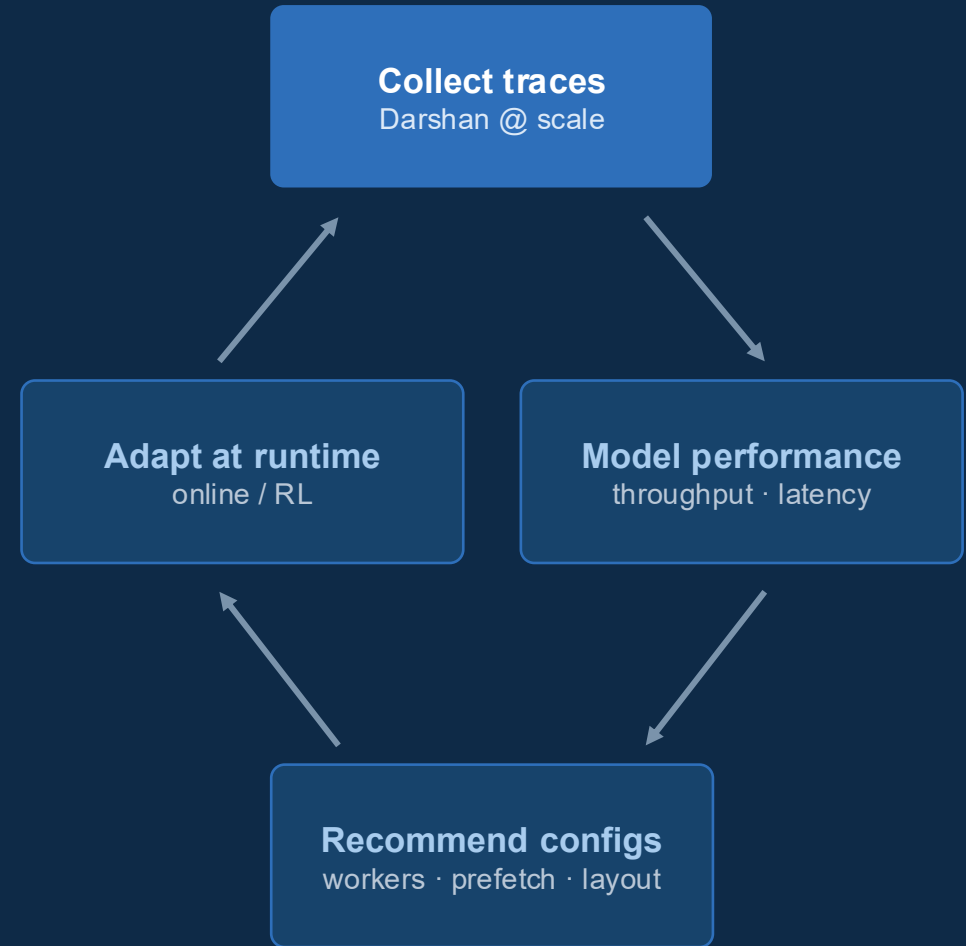


Scale up the traced runs

the worker-parallelism sweep and production-scale GN2 training

Potential ML Extension: Learning-Aware I/O Optimization

- From observational analysis to a predictive, adaptive framework
- Train ML models on the collected Darshan traces
 - workload + data format + system conditions → predicted throughput, latency, contention
- Intelligent configuration recommendation
 - prefetching strategy, parallel worker count, data-layout choices for a target workload
- Closed-loop optimization
 - monitor runtime I/O and adapt loader parameters to changing filesystem conditions
 - an element of online / reinforcement learning at scale



Summary

- Data loading is a first-order bottleneck for HEP ML training at scale
 - this project characterizes it systematically with Darshan
- Two complementary testbeds bracket the problem
 - a controlled benchmark that isolates loader parallelism
 - real ATLAS GN2 training through the salt framework on HDF5
- Instrumentation is in place, validated by reduced-scale traced training runs
- The traces answer concrete questions — access patterns, read amplification, parallel efficiency, contention — feeding the loader × format comparison and, later, a learned optimization layer

Next Steps

- Run the loader-parallelism sweep and the first large-scale traced GN2 trainings
- Broaden the comparison
 - the ROOT-based loader, and the RNTuple, NPZ, and CSV formats
- Mine the traces: where does loading time go?
 - access patterns, hotspots, and contention on shared storage
- Distill the findings into concrete data-pipeline guidance for HEP training
- Begin the ML extension — predicting I/O performance from workload and configuration