The Case of the Elusive Application Performance on Production GPU Supercomputers

Cunyang Wei
Department of Computer Science
University of Maryland
College Park, Maryland 20742 USA
cunyang@umd.edu

Keshav Pradeep
Department of Computer Science
University of Maryland
College Park, Maryland 20742 USA
keshprad@umd.edu

Abhinav Bhatele
Department of Computer Science
University of Maryland
College Park, Maryland 20742 USA
bhatele@cs.umd.edu

***Abstract*—Modern HPC facilities increasingly rely on GPU-accelerated clusters to drive both scientific computing and AI workloads. Performance variability is a critical issue in these systems, undermining efficiency and performance reproducibility. While prior studies have extensively analyzed variability in CPU-centric supercomputers, similar large-scale investigations on GPU clusters are lacking. To address this gap, we set up a longitudinal experiment on two state-of-the-art GPU-based supercomputers: NERSC's Perlmutter and ORNL's Frontier. We benchmark several representative HPC and AI applications and collect detailed performance data including network counters, profiling output, and job scheduler logs. We analyze this data to identify the impact of compute performance variations, allocated node topology, and network conditions on the overall runtime variability. We also use a machine learning based approach to identify potential correlations between these factors, and to forecast performance variability. We provide actionable insights for both system administrators and users to mitigate or predict performance variations in GPU-accelerated HPC environments.**

***Index Terms*—performance variability, GPGPUs, dragonfly network, AI workloads**

I. Introduction

Performance variability is a critical concern in modern HPC systems. Large-scale supercomputers run tightly-coupled parallel applications, so even minor slowdowns on one node can cause cascading delays across an entire job. The impact of performance variability extends to AI training workloads, which increasingly rely on HPC platforms [1]–[4]. Large-scale training jobs with synchronous distributed algorithms are particularly sensitive to delays in network communication. Many system components, especially the network, can suffer from transient performance degradation under significant load [5], [6], which can slow down training progress.

Researchers have extensively investigated performance variability in HPC environments, primarily focusing on traditional CPU-based supercomputers. Previous work has investigated issues such as operating system noise and jitter introduced by background daemons [7]. Researchers have also analyzed interference from co-scheduled jobs on shared resources, showing that a job's performance can suffer due to network contention caused by neighboring jobs [6]. Manufacturing defects between nominally identical nodes have also been shown to introduce measurable run-to-run performance differences [8]. However, with flagship supercomputers now heavily reliant on GPUs for computational power, our understanding of variability must extend to address these GPU systems.

It is important to note that while network-induced performance variability has been well-studied on CPU-based systems, GPU-enabled applications exhibit different network behaviors. GPUs provide significantly higher compute power than CPUs, which allows applications to scale to larger problem sizes per process when ported from CPU to GPU. The increase in problem size can lead to larger message sizes in communication. Additionally, traditional HPC workloads typically involve message sizes in the KB range, while deep learning training tasks can have message sizes ranging from MB to GB, causing network performance to shift from latency-dominated to bandwidth-dominated. These new characteristics in GPU systems necessitate that we must re-examine computation-communication relationships, analyze the causes of performance variability, and build predictive models to help mitigate performance variability.

In this work, we conducted extensive experiments on two flagship GPU-based supercomputers: Perlmutter at NERSC and Frontier at OLCF. To the best of our knowledge, there are no comprehensive large-scale studies that systematically analyze how network behaviors cause performance variability in GPU-based supercomputers. We repeatedly ran a suite of representative workloads to probe the systems' performance under real-world conditions. These workloads include MPI-based proxy and production applications, as well as distributed deep learning training workloads that rely on NVIDIA's NCCL and AMD's RCCL for GPU communication. By covering both MPI and NCCL/RCCL, our experiments stress the network in ways typical of traditional HPC applications and modern AI workloads, respectively. As shown in Figure 1, we observed up to 1.4× and 1.3× variability for nanoGPT and AMG2023 on Perlmutter, respectively. On Frontier, DeepCAM and MILC exhibited up to 2.6× and 1.8× variability, with occasional outliers reaching 3.2×. We removed outliers from the figures to avoid distorting the broader distribution.

For each experimental run, we captured comprehensive performance metrics including application runtime, detailed MPI or NCCL/RCCL routine profiles, and low-level network performance counters. We also recorded job scheduler logs to

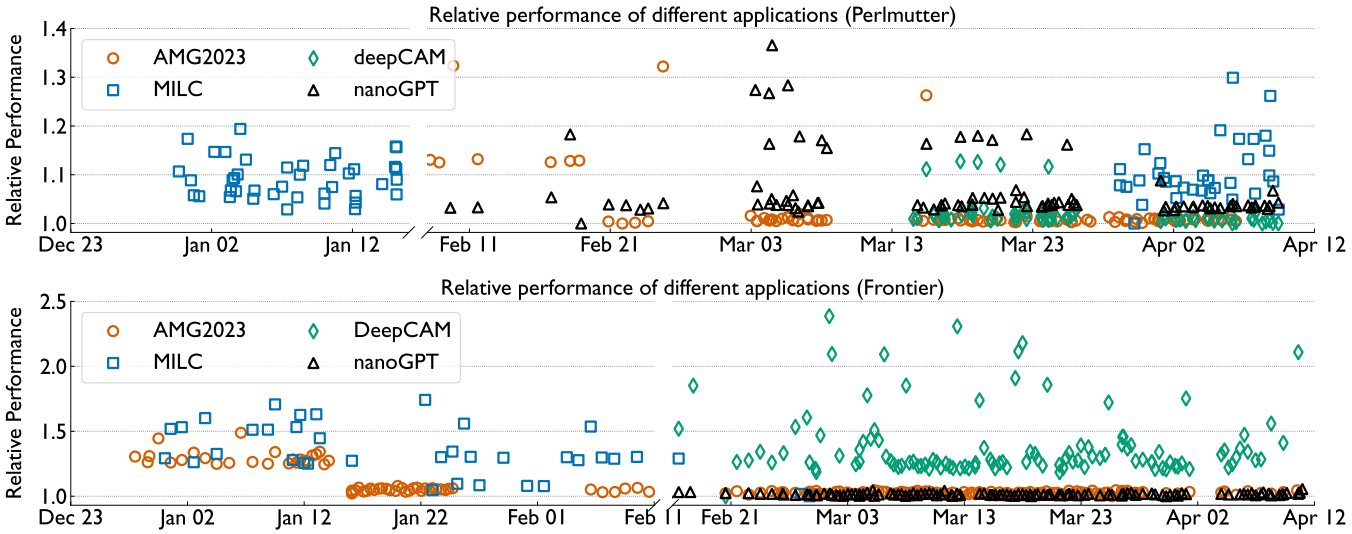

Fig. 1. Plots showing variability in performance of four different HPC and AI applications relative to their respective best observed execution times over a period of four months in 2024–2025 (jobs run on 64 nodes of Perlmutter and Frontier).

track node allocations, placement topology, and background system load. By correlating these diverse data sources, we identify when and how performance variations occur and trace their relationship to network conditions and system-wide resource contention.

Furthermore, we trained machine learning models to predict performance variations based on collected system metrics and runtime characteristics. Our models achieved high prediction accuracy on both Perlmutter and Frontier. Notably, the models maintained high accuracy even for applications with limited training data, demonstrating their generalizability.

In summary, this work makes the following contributions:

- We present a longitudinal study on two major GPU-based supercomputers (Perlmutter and Frontier), yielding a comprehensive dataset of performance measurements across diverse applications and system states. To our knowledge, this is the first study to observe and quantify performance variability on GPU-accelerated HPC systems over a significantly extended period of time and large scale.
- We summarize several key observations, providing an in-depth analysis of the inherent differences between system hardware and the impacts of concurrent jobs, job placement, and network conditions on performance.
- We use machine learning methods to both identify critical metrics that influence performance variability and predict performance variations based on system status across traditional HPC workloads and deep learning applications.
- Based on our findings, we provide insights for system administrators and users to predict and mitigate performance variations.

II. Background and Motivation

Below, we describe the HPE Cray EX system architecture used in both Perlmutter and Frontier, the two supercomputers evaluated in this paper, and discuss potential sources of performance variability on HPC systems.

A. Dragonfly-based HPE Cray EX System

Dragonfly is a high-radix, low-diameter network topology [9]. It consists of groups of routers connected in all-to-all connections. Within each group, each router (switch) is directly connected to every other router, and the groups are interconnected by high-bandwidth global links to form a two-level hierarchy. The dragonfly design employs adaptive routing techniques to efficiently distribute network traffic and congestion management mechanisms to prevent bottlenecks. This architecture ensures consistently low communication latency, even under varying workloads, and is particularly well suited for HPC applications.

The HPE Cray EX (Shasta) series implements a dragonfly-based interconnect called Slingshot [10], which is a high-performance network designed for exascale scalability. Slingshot's switching ASIC (codename Rosetta) is a high-radix 64-port switch providing 12.8 Tb/s of bisection bandwidth and supporting link speeds of 100 or 200 Gbps per port. These switches are arranged in a two-level dragonfly topology. A typical Cray EX configuration uses 32 Rosetta switches per group, each fully interconnected with the others in that group, and dedicating up to 16 ports on each switch for connections to other groups. This yields a low-diameter network (max three hops) that can scale to over 250,000 endpoints while maintaining full bandwidth between groups.

B. Sources of Performance Variability

Even on well-designed HPC systems, runtime performance can vary from run to run due to a variety of factors. The primary sources of performance variability include differences or inconsistencies in hardware, contention in the network, inefficiencies in job scheduling and task placement, and fluctuations in the software stack or system software behavior.

TABLE I
APPLICATIONS AND THEIR RESPECTIVE INPUT PARAMETERS ON 64 NODES OF PERLMUTTER AND FRONTIER

Application	Machine	Key input parameters	Number of jobs
AMG2023	Perlmutter	-P 4 8 8 -n 128 64 64 -problem 1	104
AMG2023	Frontier	-P 8 8 8 -n 128 64 64 -problem 1	168
MILC	Perlmutter	nx 40 ny 160 nz 320 nt 320 node_geometry 1 4 8 8	78
MILC	Frontier	nx 80 ny 160 nz 320 nt 320 node_geometry 2 4 8 8	38
DeepCAM	Perlmutter	-max_epochs 8 -local_batch_size 2 -data_format dali-es-gpu	67
DeepCAM	Frontier	-max_epochs 4 -local_batch_size 2 -data_format hdf5	109
nanoGPT	Perlmutter	model_name gpt2 iters 30 model_size 20B batch_size 8 block_size 512 grad_acc_steps 256	94
nanoGPT	Frontier	model_name gpt2 iters 30 model_size 20B batch_size 8 block_size 512 grad_acc_steps 512	103

HPC applications often rely on a high-performance interconnect to communicate between nodes. When multiple jobs share the network infrastructure, they can contend for bandwidth, causing the communication speed of one job to slow down unpredictably due to traffic from another job. This network contention is a major cause of performance variability on large machines [6], [11]–[14]. A study on a Cray XC system [5] showed that when an application had to share network links with other jobs, its runtime could be up to 3× longer than in runs with no interference.

There are studies showing that sub-optimal node placements [15]–[18] also contribute to performance variability. If an MPI job is assigned to nodes that are scattered rather than contiguous, its messages may traverse more hops, increasing network latency.

Furthermore, operating system behavior and runtime system activity can introduce jitter in an application's performance. This phenomenon is known as OS jitter or OS noise, and it has long been recognized as a cause of performance variability in tightly coupled parallel programs [7], [19]–[21].

C. Motivation for This Study

As discussed in the previous section, most prior work on performance variability has focused on CPU-based HPC systems. The causes of variability on GPU clusters have not been analyzed systematically. Prior studies suggest that differences in GPU compute performance can affect application variability [8], [22], but much of that evidence comes from single node jobs, leaving the impact on large parallel runs unclear. CPU-based and GPU-based systems also differ in a fundamental way: GPU nodes deliver far more computational power per node, while network bandwidth per node typically does not scale up at the same rate. In practice, system costs limit how much network bandwidth can be added per unit of compute. This widens the flop/s-to-bytes gap and can make congestion more likely. As a result, even when an application uses the same number of nodes, its compute behavior and communication patterns can shift when moving from CPU nodes to GPU nodes. These system-level differences also mean that conclusions from previous studies do not always carry over to large-scale GPU systems.

AI training workloads push this shift further. They often exchange much larger messages than traditional HPC codes. This shift causes the network to become a bottleneck, moving communication performance from being latency-dominated to bandwidth-dominated. These differences motivate a fresh look at the relationship between computation, communication, and variability on modern GPU-based supercomputers.

III. EXPERIMENTAL SETUP

In this section, we provide details of the machines used, and applications and benchmarks employed in our evaluation.

A. Machines Used

Perlmutter is an HPE Cray EX system at NERSC with 4,864 nodes, including 1,792 GPU-accelerated nodes and 3,072 CPU-only nodes. In this study, we only use the GPU nodes. Each GPU node contains one 64-core AMD EPYC 7763 Milan CPU, and four NVIDIA A100 GPUs (40GB or 80GB HBM2 per GPU). Perlmutter uses the HPE Slingshot-11 high-speed interconnect, which is a 3-hop dragonfly network. The Slingshot network interfaces (25 GB/s each, via PCIe 4.0) are HPE Cray Cassini NICs. GPU nodes have four NICs, one per GPU (for 100 GB/s aggregate injection bandwidth per node).

Frontier, the first Exascale supercomputer, is also an HPE Cray EX system at OLCF. It consists of 9,408 compute nodes, each with one 64-core AMD EPYC Trento CPU and four AMD Instinct MI250X GPUs. Each MI250X contains two GPU dies (GCDs) for eight total GCD devices per node. Each GCD has 64 GB of HBM2e high-bandwidth memory. Frontier also uses the HPE Slingshot-11 network. Each compute node is equipped with four Slingshot NICs (25 GB/s each) for a total of 100 GB/s bidirectional bandwidth per node.

B. Applications and Their Inputs

We use two representative HPC applications, *AMG2023* and *MILC*, and two AI training workloads, *DeepCAM* and *nanoGPT* for our experiments. These applications were chosen to represent distinct and diverse computational patterns and communication requirements, including the use of MPI and NCCL/RCCL [23], [24] respectively. We configure the applications and choose their input problems based on the specifications provided in the NERSC-10 benchmark suite [25] and the OLCF-6 benchmark [26]. Additionally, we confirmed with the developers of AMG2023, nanoGPT, and MILC, that our setup aligns with configurations in production runs.

TABLE II
SUBSET OF NIC COUNTERS RECORDED FOR PERFORMANCE ANALYSIS.

Counter	Simplified Description
rh:sct_timeouts	Response timeouts that trigger retries
rh:spt_timeouts	Retry handler: packet loss events
hni_rx_paused_0/1	Cycles where receive path is paused (0 for request and 1 for response)
hni_tx_paused_0/1	Cycles where transmit path is paused (0 for request and 1 for response)
lpe_net_match_request_0	Messages matched on the request list
lpe_net_match_priority_0	Messages matched on the default priority list
lpe_net_match_overflow_0	Messages matched in overflow buffer
atu_cache_hit_base_page_size_0	Number of cache hits observed on the Base Page Size.
atu_cache_hit_derivative1_page_size_0	Number of cache hits observed on the Derivative 1 Page Size.
parbs_tarb_pi_posted_pkts	Number of PCIe packets transferred using the posted path (e.g. writes)
parbs_tarb_pi_posted_blocked_cnt	Number of cycles in which the posted path is blocked
parbs_tarb_pi_non_posted_blocked_cnt	Number of cycles in which the non-posted path is blocked

Table I summarizes the input parameters for each application. Note that each node on Frontier has double the GCDs and significantly more HBM2e memory. Therefore, the scale of computation is slightly different from that on Perlmutter.

AMG2023 is a parallel algebraic multigrid solver, based on the Hypre library [27], designed for linear systems arising from problems on unstructured grids [28]. We use Problem 1, which solves for a 3D diffusion problem on a cuboid with a 27-point stencil. We modified the main solver to repeat the solve 500 times in order to capture longer execution times and observe temporal variability more effectively.

MILC: The MILC (MIMD Lattice Computation) code is a widely used application for studying quantum chromodynamics (QCD) [29]. MILC exhibits uniform computation and communication patterns per simulation step, making it a popular benchmark for evaluating performance on large-scale HPC systems. Our experiments use the `su3_rhmd_hisq` application, which performs gauge generation using the rational hybrid molecular dynamics (RHMD) algorithm with HISQ quarks, accelerated via the QUDA library [30]–[33].

DeepCAM is an exascale-class deep learning application designed for climate analytics, using convolutional neural networks to segment extreme weather patterns [34]. It implements a convolutional encoder-decoder architecture trained on CAM5 (Community Atmosphere Model) climate simulation data, with TECA-generated segmentation masks for labeling [35], [36].

The input dataset consists of 121,266 training and 15,158 testing samples, with a total size of 8.8TB [35]. On Perlmutter, the dataset is stored in `.npy` (NumPy) format, which is compatible with NVIDIA's Data Loading Library (DALI) for efficient preprocessing and loading. On Frontier, the dataset is stored in `.hdf5` (HDF5) format, which allows optimized access patterns suited for the AMD-based system.

nanoGPT is an optimized implementation of GPT. Our reference implementation leverages AxoNN, which is a highly scalable distributed training framework [1], [37]. For the parallelization parameters of AxoNN, we set $G_{intra\_z} = 16$, $G_{intra\_x} = 1$, and $G_{intra\_y} = 1$.

C. Compute and Communication Micro-benchmarks

Before each application runs in a job, we first execute two benchmarks – GEMM and All-reduce to probe the raw performance of our compute and communication sub-systems. This approach allows us to establish inherent performance characteristics and detect variability in fundamental operations. Specifically, we benchmark the FP16 GEMM performance of each GPU to assess the peak performance and compare each GPU's performance to that of other GPUs. Additionally, we benchmark collective communication performance using two paradigms: `MPI_Allreduce` for traditional HPC applications and `NCCL/RCCL_Allreduce` for distributed AI training, which help us understand the network's behavior.

IV. DESCRIPTION OF COLLECTED DATA

Understanding performance variability in HPC systems requires gathering data from diverse sources that capture both system-wide conditions and job-specific behaviors. To diagnose such variability rigorously, we collect multiple layers of performance data: (a) Network hardware performance counters that record low-level interconnect events, indicating systemic network conditions, (b) Data from profiling tools that capture detailed information about communication and computation routines, and (c) Slurm job logs that detail how and where each job ran relative to others [38], [39].

A. Network Hardware Performance Counters

To gain insight into the network's behavior during job execution, we leverage the NIC's (Cray Cassini) network hardware performance counters [40], which record low-level events and performance metrics. The NIC counter data is recorded during `MPI_Init` (job start) and `MPI_Finalize` (job end) for each process. Even AI applications that do not use MPI can have their network information recorded through these counters, as all inter-node traffic must pass through the NIC regardless of the communication library used. Table II shows a subset of the NIC counters that we collected for performance analysis.

B. Profiling Data from Performance Tools

We use `mpiP` to profile MPI applications, and `PyTorch Profiler` to profile PyTorch-based deep learning workloads.

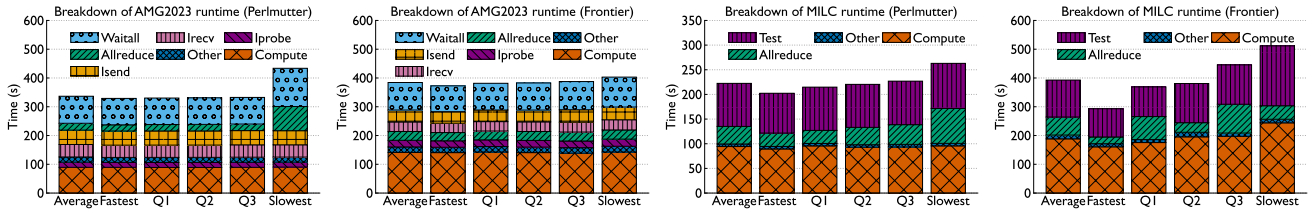

Fig. 2. Execution time breakdowns for AMG2023 and MILC on Perlmutter and Frontier.

`mpiP` collects and aggregates performance metrics for MPI routines, such as cumulative execution times, and average durations per call [41]. In particular, we use `MPI_Pcontrol` to capture MPI data exclusively from the main execution loop, thereby excluding initialization and I/O overheads.

The `PyTorch Profiler` records both CPU and GPU operations, capturing fine-grained metrics such as operation execution times, memory consumption, and compute-communication overlap [42]. We use HTA [43] to analyze the output from the profiler. Currently, HTA does not support detailed breakdowns for RCCL, so we modified the HTA source code to enable this capability[1]. Again, we isolate the main loop to minimize interference from initialization.

C. Job Placement and Concurrent Jobs Data

We analyze each job's placement and jobs running concurrently with it by using Slurm logs via `sacct` [39]. From these logs, we map each job's nodes to their dragonfly groups and then perform a topology-aware placement analysis. We also identify other jobs running concurrently with each job. By parsing the Slurm logs, we obtain for other concurrent jobs: (a) the list of nodes they ran on and (b) their start and end times, which we then cross-reference against our job's records. If other jobs run at the same time as our job, we flag them as concurrent jobs that could introduce interference.

V. Overview of Application Variability

In this section, we analyze each application's performance to understand the root causes of performance variability. In addition, we analyze performance variation in the GEMM benchmark to understand the impact of GPU hardware on application performance. We have excluded several extreme outliers from the figures presented in this section, as their inclusion would distort the broader distribution. These outliers are included in subsequent sections.

A. Characterization of Individual Applications

We collect MPI profiles via mpiP [41] to characterize the HPC applications, and use the PyTorch Profiler [42] to profile AI applications, focusing on time spent in computation versus communication and on identifying the most variable routines. Also, since profiling the entire execution of AI applications generates hundreds of GB of logs per run, we limit profiling to five iterations after ten warmup iterations for each run. As a result, the variability depicted in this section might differ from the overall variability shown in Figure 1. Our focus here is primarily on dissecting the sources and composition of performance variability.

We observed a significant reduction in AMG2023 variability following a routine maintenance on Frontier on January 14, 2025, as shown in Figure 1. Starting January 14, Frontier runs used an upgraded Slingshot Host Software 11.0.2, which fixed a known performance regression in libfabric 1.20.1. Since both NCCL and MPI traffic goes over Slingshot, we believe that the observed reduction in performance variability after January 14 can be attributed to this upgrade. To ensure consistency in our analysis, the results presented for AMG2023 on Frontier are based on data after January 14.

AMG2023: The first two plots in Figure 2 demonstrate the detailed breakdown of the time spent in AMG into different MPI routines on Perlmutter and Frontier. Here, "Fastest" and "Slowest" represent the runs with the minimum and maximum total runtime, respectively. "Average" shows the sum of the average time spent in each routine category across all runs, while Q1, Q2, and Q3 correspond to the first, second, and third quartiles of the total runtime distribution, showing the breakdown for runs at those specific quartiles. The most time-consuming routines are `Waitall`, `Isend`, `Irecv`, and `Allreduce`.

Notably, there is no significant fluctuation in compute time. Instead, most of the variations arise in MPI routines. On average, at 64 nodes, approximately 74% and 84% of the total runtime is devoted to MPI routines on Perlmutter and Frontier, respectively. In the slowest runs on Perlmutter, the communication phase requires about 40% more time than in the fastest case, underscoring the strong impact of communication variability on AMG2023's overall performance.

MILC: Compared with AMG2023, MILC exhibits several distinctive characteristics. Figures 2 (right two plots) illustrate the distribution of compute and communication time at 64 nodes. In this case, we observe minor fluctuations in compute time, but MPI still remains the primary source of variability. On average, at 64 nodes, communication accounts for 56% and 53% of the total runtime on Perlmutter and Frontier, respectively. Breaking down the MPI overhead reveals that the `Allreduce` and `Test` routines dominate. In the slowest runs, MILC's total communication time can be up to 50% (Perlmutter) and 122% (Frontier) higher than in the fastest

[1]https://github.com/hpcgroup/perf-variability-ipdps26

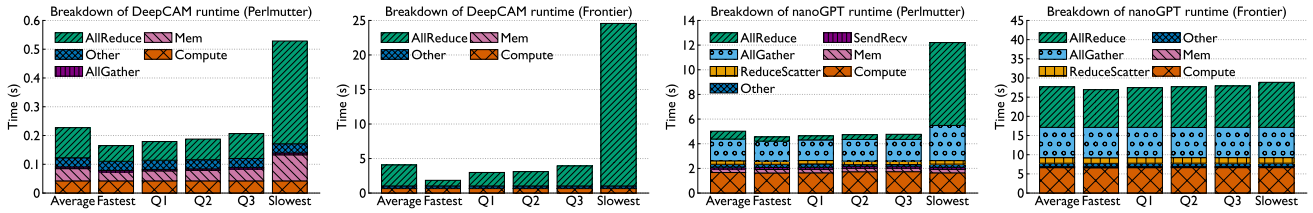

Fig. 3. Execution time breakdowns for DeepCAM and nanoGPT on Perlmutter and Frontier.

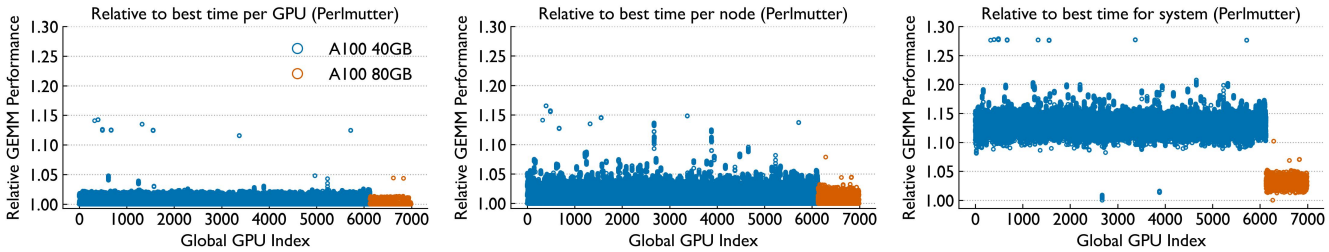

Fig. 4. Variability in relative GEMM performance on Perlmutter. Each plot shows GEMM times normalized to the best times across different resource granularities: individual GPU (left), individual node (middle), and system-wide (right).

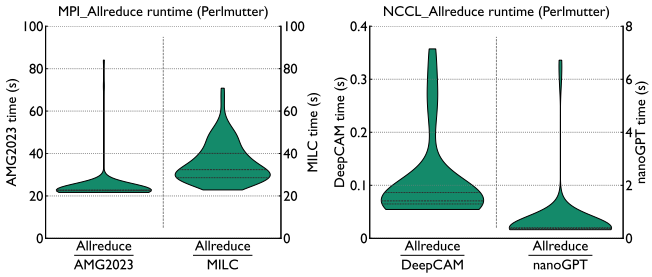

Fig. 5. Distribution of Allreduce on Perlmutter.

runs. Note that, in contrast to AMG, MILC does suffer from significant variability on Frontier also.

DeepCAM: Figures 3 (left two plots) present the breakdown of DeepCAM's execution time. In these figures, "Mem" includes memory transfers between the host and devices and between devices. We observe no major fluctuations in the `Compute` and `Other` categories. The performance variability in DeepCAM is significantly worse than that of the HPC applications. `Allreduce` is responsible for most of the performance variability, and in the slowest runs, it can take up to four times longer (Perlmutter) or up to 24 times longer (Frontier) than in the fastest runs.

nanoGPT: Figures 3 (right two plots) illustrate nanoGPT's performance breakdown. Although `Allgather` constitutes the largest portion of communication time in the average case on Perlmutter, the greatest variability in overall performance can be attributed to `Allreduce`, which in the slowest cases can be three times slower than the fastest observed runs. On Frontier, `Allreduce` exhibits consistent performance across runs, showing no significant difference between the fastest and slowest cases.

Allreduce distribution: As mentioned earlier, `Allreduce` exhibits the greatest variability across all applications. Therefore, Figure 5 presents the `Allreduce` time distribution of all four applications on Perlmutter. The results show that some runs are significantly slower than typical execution times.

Takeaway 1

Performance variability arises primarily due to slowdowns in collective communication, in particular, `Allreduce`, `Test`, and `Waitall` routines. Some routines exhibit long-tail effects.

B. Characterization of GEMM Variability

We wanted to separate out compute variability issues from communication variability. To gain insights into both the performance variability of individual GPUs over time and performance differences across GPUs, we conducted matrix multiply (FP16 GEMMs) benchmarking on Perlmutter and Frontier. On Perlmutter, we distinguish between two types of A100 GPUs by node ID: nodes with $ID > 8000$ are equipped with 80 GB of DRAM per GPU (shown in orange in Figure 4), and $ID < 8000$ have 40 GB (shown in blue).

Figure 4 (left) shows the ratio of GEMM execution time to the best-observed time on the same GPU for various runs in our dataset. We use this metric to evaluate whether each GPU's performance exhibits significant fluctuation over different runs. The results show that all GPUs have some variability in performance (within a 2.5% window) compared to their best recorded execution time. There are a small number of samples outside this range (only 0.066%), indicating that individual GPU performance remains relatively stable.

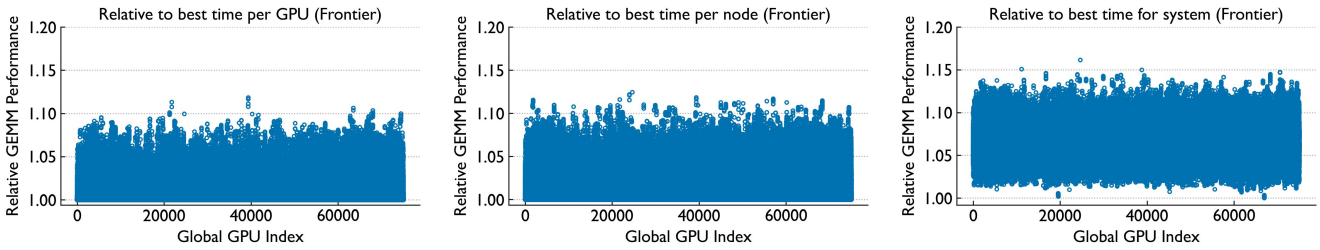

Fig. 6. Variability in relative GEMM performance on Frontier. Each plot shows GEMM times normalized to the best times across different resource granularities: individual GPU (left), individual node (middle), and system-wide (right).

Figure 4 (middle) further examines the performance variability within each node. We normalize GEMM performance to the best performance observed on any GPU within the node. This result reveals that even among the four GPUs within a single node, performance differences can reach 10%, and in the worst cases, 17%.

Finally, we normalize each GPU's performance to the best observed GEMM performance on the entire system (A100 40GB and A100 80GB normalized by the best performance of A100 40GB and A100 80GB, respectively). As illustrated in Figure 4 (right), the system-wide variability is considerable, reaching up to 28% even among GPUs of the same model from the same vendor. Notably, the A100 80GB GPUs exhibit lower variability compared to the A100 40GB. After removing outliers, the system-wide variability is about 12% for A100 40GB and 5% for A100 80GB. In addition to lower variability, the A100 80GB GPUs deliver approximately 7% higher average GEMM performance compared to the A100 40GB GPUs.

Similarly, as illustrated in Figure 6, we evaluated the GEMM performance across all GCDs on Frontier. Figure 6 (left) shows that most GPUs on Frontier exhibit performance variations of up to 12% when normalized to the best performance of the same GPU. This suggests that the MI250X GPUs on Frontier demonstrate more performance variability compared to the A100 GPUs on Perlmutter. The middle and right plots of Figure 6 further illustrate intra-node and system-wide performance variability. Compared to Perlmutter, Frontier exhibits greater overall variability, but with fewer extreme outliers. System-wide performance variations reach up to 15%, while the extreme outliers show up to 16% variability.

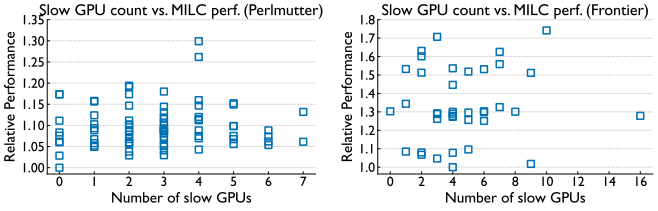

Fig. 7. Impact of the number of slow GPUs in the job allocation on MILC runtime (left: Perlmutter, right: Frontier)

In large-scale parallel workloads, collective communications occur frequently, and overall performance often hinges on the slowest GPU. As shown in Figure 7, we examined the correlation between application runtime and the number of GPUs within the job's allocation that fall into the system-wide slowest 1% (based on GEMM performance). Values of Spearman's correlation coefficient are 0.07 and 0.08 on Perlmutter and Frontier, respectively. We observed the same weak correlation when extending this to the slowest 10% and 30% of GPUs. This suggests that performance variability likely stems from other factors, such as network congestion, rather than from slower GPUs.

Takeaway 2

While single-GPU performance remains relatively stable over time (especially on Perlmutter), there is notable variability across different GPUs. Moreover, GEMM variability tends to be higher on Frontier, though it exhibits fewer extreme outliers compared to Perlmutter. Despite this, GPU-level variability does not correlate with application-level variability.

VI. Impact of Job Placement

We now analyze the impact of job placement on performance variability. Our analysis covers two key aspects: the influence of the number of assigned dragonfly groups on performance variability, and the correlation between concurrent jobs activity and performance degradation.

A. Allocation Spread across Dragonfly Groups

One possible hypothesis for the observed performance variability is that allocating nodes across a larger number of dragonfly groups increases the number of network hops, thereby impacting overall network performance. To investigate this, we measured the performance of both HPC and AI applications on 64 nodes of Frontier and Perlmutter over an extended period, recording the number of dragonfly groups used in each job's node allocation.

As shown in Figure 8, the runtime of nanoGPT and DeepCAM do not correlate with the number of dragonfly groups on both Perlmutter and Frontier, with Spearman's correlation coefficients of 0.33 and 0.08, respectively. In particular, on Frontier, there is no significant change in performance between being assigned to one group and being assigned to 64 dragonfly groups. Other applications show similar results.

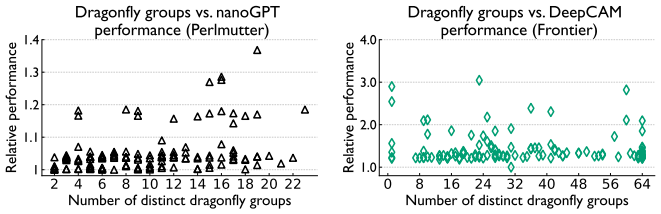

Fig. 8. Impact of number of dragonfly groups a job is allocated to on application performance (left: nanoGPT on Perlmutter, right: DeepCAM on Frontier)

This finding suggests that the dragonfly group count does not impact the overall performance of applications. This suggests the robustness of the indirect UGAL routing algorithm [9], which allows Perlmutter and Frontier to scale efficiently without experiencing performance loss associated with multi-group topology scaling.

Takeaway 3

Dragonfly topology implementations on both Perlmutter and Frontier maintain high performance and scalability, even when computational tasks are allocated to a large number of dragonfly groups.

B. Impact of Concurrently Running Jobs

Modern HPC systems often use high-radix networks such as dragonfly to provide fast, large-scale communication. Even with these designs, we observe performance variations when multiple jobs share the system. As discussed in Section III, our analysis primarily focuses on multi-node HPC jobs. To pinpoint the factors influencing performance variability, we first identify concurrent jobs capable of impacting inter-node communication, which we define as *relevant jobs*. In particular, concurrent jobs spanning multiple dragonfly groups are considered *relevant*, even if they do not directly share any groups with our application, due to their potential to introduce cross-group congestion via dynamic routing [9]. Conversely, we exclude single-node jobs and multi-node jobs fully contained within a single dragonfly group, provided that this group does not overlap with any group used by our application, since these jobs do not generate cross-group network interference.

We first investigate whether the total allocated nodes from all relevant jobs correlate with the runtime of our application. Note that the number of concurrent nodes we obtained is the sum over the time period our job was running, so it is possible that two jobs are mapped to the same nodes, meaning the number of nodes in our figures may exceed the total number of nodes in the system. Figure 9 illustrates the results for both Perlmutter and Frontier, with Spearman's correlation coefficients of 0.03 and 0.39 respectively. This shows no clear relationship between aggregated system-wide node usage and runtime on Perlmutter, while Frontier exhibits weak correlation. This outcome indicates that aggregate node allocation alone fails to explain performance variations. This is likely due to the noise introduced by low communication demand tasks obscuring the correlation between high communication demand jobs and performance variability.

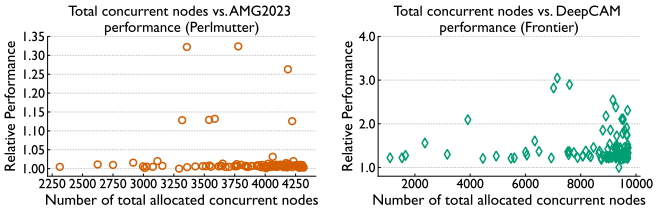

Fig. 9. Impact of total number of nodes allocated to relevant jobs on performance variability (left: AMG2023 on Perlmutter, right: DeepCAM on Frontier)

To address this, we perform a refined analysis aimed at identifying a subset of *Top Users* who can likely affect application runtime. During each run of our application, we sum the number of nodes that each user allocates for *relevant jobs*. We call this the *Relevant User Node Count*. We then compute the correlation coefficient between this node count and our observed runtime, termed the *Relevant User Correlation*. We designate a user as a *Top User* if the *Relevant User Correlation* exceeds 0.3 and that user has requested more than 32 nodes concurrently. This definition ensures we capture users whose workload patterns directly align with observed variability and who periodically generate significant network loads.

As shown in Figure 10, our analysis reveals a distinct pattern: whenever the concurrent node allocation by these *Top Users* crosses a certain threshold, our application's runtime consistently increases. For example, on Perlmutter, we observe performance degradation of at least 7% for AMG2023 when *Top Users* collectively occupy over 300 nodes. Although this relationship is not strictly linear, significant node usage by *Top Users* consistently drives longer runtime, with Spearman's correlation coefficients of 0.55 and 0.60 for Perlmutter and Frontier respectively. That is, optimal performance always occurs when these *Top Users* do not acquire many nodes.

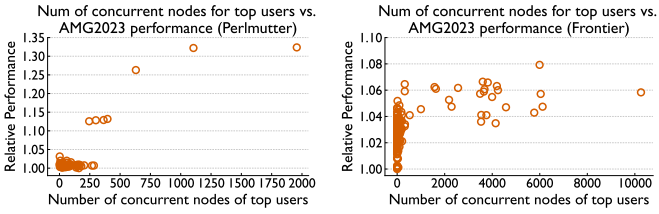

Fig. 10. Impact of number of nodes allocated to top users on performance variability of AMG2023 (left: Perlmutter, right: Frontier)

To further validate our findings, we manually investigate the specific composition of these *Top Users*. For instance, on Perlmutter, User A frequently submits multiple `vasp_gam` tasks, which is a simulation package to model atomic scale materials from first principles. Nearly all of our application

runs that show a 15% slowdown coincide with large node allocations for `vasp_gam`.

Takeaway 4

Overall system utilization alone does not explain the observed performance degradation; a few specific neighbors with high communication intensity can cause most of the performance variability.

VII. Statistical Analysis of Longitudinal Data

Next, we turn to statistical analysis and machine learning methods to investigate the key factors contributing to performance variability in HPC and AI workloads.

A. Correlating NIC Counters to Performance

First, we compute Spearman's correlation coefficients between application execution times and mean / maximum values of various NIC counters (presented in Figure 11). A positive correlation indicates that higher counter values tend to coincide with longer runtimes (and thus degraded performance), whereas negative correlations imply that larger counter values correlate with shorter execution times or better performance. Note that this heatmap does not include all available NIC counters. We have excluded those that exhibited little to no correlation with application runtime. Additionally, we do not show DeepCAM due to its weak correlation with NIC counters.

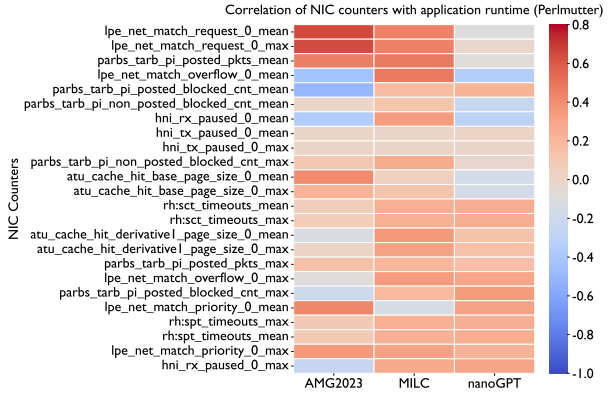

Fig. 11. Heatmap showing correlation between application runtime and mean & max values of selected NIC counters.

We observe that `rh:{sct/spt}_timeouts` show positive correlations with several applications. These counters reflect retry events and response timeouts triggered by packet loss. Their strong association with longer runtime aligns with intuition: frequent retries and cancellations disrupt steady data flow, prolonging execution time.

Another noteworthy group includes `hni_rx_paused_0` and `hni_tx_paused_0`, which track backpressure on receive or transmit paths. They likewise exhibit positive correlations for MILC, suggesting that prolonged pauses tend to degrade overall performance. However, our analysis of `lpe_net_match_*` and `atu_cache_hit_*` counters contradicts our initial assumption that improved message matching efficiency and higher ATU cache hit rates should lead to better performance. We observe similar results for applications on Frontier. We believe that analyzing NIC counters in isolation may not fully capture the complexities of variability. This suggests that we need to uncover more intricate underlying relationships.

Takeaway 5

While NIC counters show some correlation with performance variability, they do not fully capture the complexity involved. Other models are needed to identify potential relationships affecting performance.

B. ML-based Performance Forecasting

In the previous sections, we explored the impact of different metrics on performance. However, real runs often show variability stemming from multiple interacting factors. To capture these sources of variability more comprehensively, we use machine learning methods to analyze the performance data. Since we only started collecting NIC counters after February 8th, the data analyzed in this section is a subset of what is described in Table I.

Methodology of Performance Prediction: As described in Section IV-A, we collected logs from multiple runs of each application and extracted several metrics as shown in Table III. We use these metrics to train a model to predict the application's runtime.

TABLE III
Description of features in the dataset.

Name	Explanation	Feature Count
Application Name	Using one-hot encoding.	1
Placement	The number of dragonfly groups assigned to a job.	1
GEMM	Average, minimum, and maximum compute times on all nodes.	3
MPI Allreduce	Proxy for network throughput. Test size: 1 KB, 2 KB, …, 1 MB.	11
NCCL Allreduce	Proxy for network throughput. Test size: 16 MB, 32 MB, …, 2 GB.	8
NIC Counters	Average, minimum, and maximum values from each node's NICs. Important counters are described in Table II.	29*3

We use the XGBoost (eXtreme Gradient Boosting) regression model [44] as our primary method. XGBoost handles mixed data types and performs well in predicting the performance of parallel programs [45]. Although we evaluated several models, XGBoost predictions were most accurate. Thus, we focus on it in our analysis. XGBoost is a gradient boosting technique that trains decision trees in sequence. Each tree attempts to correct the errors of earlier trees, and the final prediction emerges from the sum of all trees. It also provides feature importance estimates by tracking how many times each feature reduces impurity in the trees. Feature importances can be used to identify the counters that contribute most to runtime variability. During the training process, 10% of the data is kept as a test dataset, while the other 90% is used for training.

Evaluation Metrics: We evaluate the model's performance using two different metrics: Mean Absolute Percentage Error

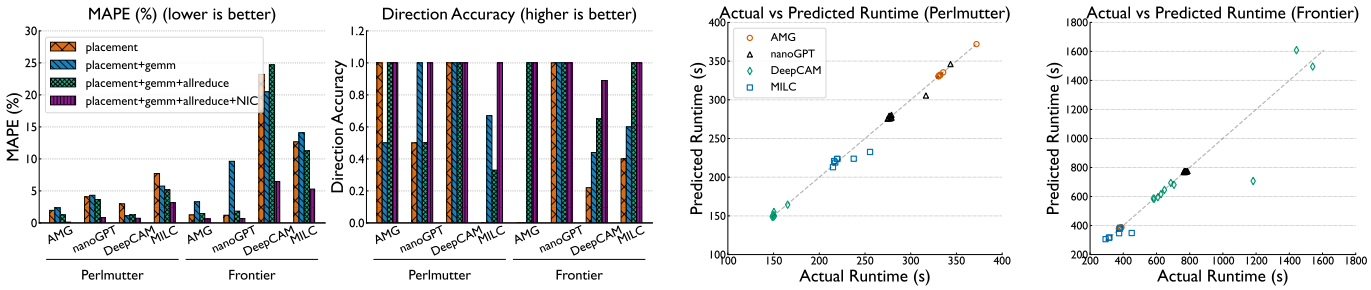

Fig. 12. Analysis of XGBoost runtime predictions on Perlmutter and Frontier. (a) Mean Absolute Percentage Error (MAPE) for each application on both systems across incremental feature combinations; (b) Direction Accuracy (DA) for each application across identical feature sets. (c) Actual vs predicted runtime using placement, GEMM, Allreduce, and NIC counters features (Perlmutter); (d) Actual vs predicted runtime with identical feature set (Frontier);

(MAPE) and Direction Accuracy (DA). MAPE measures how closely the predicted runtime matches actual observations:

$$\text{MAPE} = \frac{100}{N} \sum_{i=1}^{N} \left| \frac{y_i - \hat{y}_i}{y_i} \right|, \tag{1}$$

where y_i is the true runtime, $\hat{y}_i$ is our predicted runtime, and N is the total number of test samples.

Second, we use DA to check whether the model correctly predicted trends—specifically, whether it recognizes when performance is improving or declining compared to a previous state. To define this more concretely, let $\{y_1, y_2, \ldots, y_N\}$ be the sequence of true runtime, and let $\{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_N\}$ be the corresponding predictions. We first compute the actual change $\Delta y_i = y_{i+1} - y_i$ and predicted change $\Delta \hat{y}_i = \hat{y}_{i+1} - \hat{y}_i$. We focus on changes that exceed a certain fraction ϵ of the original runtime y_i, i.e., $\text{RelativeChange}_i = \frac{|\Delta y_i|}{|y_i|} > \epsilon$. If this is true, we check the sign consistency using: $\Delta y_i \cdot \Delta \hat{y}_i > 0$, and define the threshold-based direction accuracy as follows:

$$\text{DA}_\epsilon = \frac{\sum_{i \in \mathcal{I}_\epsilon} \mathbf{1}(\Delta y_i \cdot \Delta \hat{y}_i > 0)}{|\mathcal{I}_\epsilon|} \tag{2}$$

where $\mathcal{I}_\epsilon$ is the set of indices i for which $\text{RelativeChange}_i > \epsilon$. This metric evaluates whether the model correctly captures the broader performance trend. For instance, $\epsilon = 0.02$ means that if the runtime at step i is 300 s and then at step $i+1$ is 299 s (0.33% lower), we will not evaluate this minor variability.

Together, these two metrics provide a comprehensive view: we focus not only on the accuracy of the prediction but also on the performance trends.

Feature Importances: We start with examining the feature importances on Perlmutter and Frontier, as shown in Figure 13. On Perlmutter, `hni_rx_paused_0_mean` and `allreduce_2GB` stand out. The first metric tracks the number of cycles where traffic class 0 (request) remains paused on the receive path, suggesting that the network is pushing data more rapidly than the endpoint NIC can consume it. The Allreduce performance is a good proxy for overall network performance. Together, these features show that NIC congestion and system network congestion are strong predictors of performance variation on Perlmutter.

On Frontier, `lpe_net_match_request_0_mean`, `atu_cache_hit_derivative1_page_size_0_mean`, and `parbs_tarb_pi_non_posted_blocked_cnt_mean` dominate the prediction. The first metric tracks requests matched on software endpoints, the second measures cache hits on Derivative 1 Page Size, and the third logs cycles in which the non-posted path (for example, reads) is blocked. All three describe data movement across the system, and they reveal that uneven load with respect to data handling drives runtime variability on Frontier.

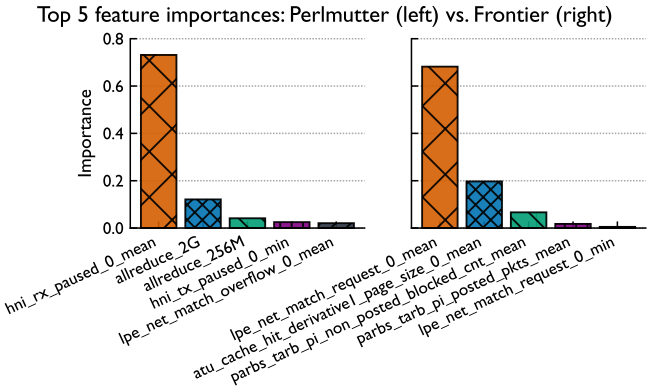

Fig. 13. Feature importances based on XGBoost models.

Forecasting Application Performance: Finally, we use the XGBoost regression model to predict application runtimes in the testing set. Figure 12 (left two plots) show the MAPE and DA for these four applications on the two machines. We use different subsets of the features to understand what data is the most important in performance predictions. When we include NIC counters in the input, the MAPE decreases significantly, especially for DeepCAM, which has greater performance variation. The DA also remains close to 1.0 when NIC counters are included. Without NIC counters, the predictions sometimes fall to chance-level performance. These results highlight the importance of NIC counters in explaining performance variability. This reveals that although static correlation analysis alone may yield limited insights (Section VII-A), leveraging NIC counter data within more sophisticated predictive models is the key to understanding performance variations.

Figure 12 (right two plots) show the predicted versus actual performance for the four applications tested in this paper. We observe that our model predicts the absolute performance for most test points reasonably well. However, some deviations in the predictions exist. We suspect this discrepancy might arise because we lack access to network router (Rosetta) counters within the system.

Additionally, we tested the performance of training separate models on data from each application. We found that the model trained on data from all applications exhibited some advantages in both DA and MAPE. We believe this is because these applications all run on the same underlying hardware, despite the different computational and communication behaviors. This shared environment creates fundamental bottlenecks such as network congestion. Models trained across multiple applications learn these common performance variation patterns more robustly than those trained on individual applications. This additional breadth helps XGBoost avoid overfitting to noise and instead learn generalizable rules.

It is worth noting that after a system update on Frontier, our MILC calculations stopped converging. We reported this issue to OLCF. As a result, we were limited to only seven MILC runs for training the Frontier prediction model, with another seven runs for testing. Even with this limited data, the model still gave good predictions, as shown in Figure 12. These findings support the idea that the model is generalizable to the system as a whole, rather than specific to individual applications. Given data from enough applications on a system, we expect that such models could eventually predict performance for unseen applications.

Takeaway 6

Traffic saturation causes Perlmutter network processing to stall, while Frontier's blocking reads and cache hits expose bottlenecks in local data movement. These findings reveal how network-driven behavior dictates performance variability on both machines.

VIII. Discussion

Based on the findings in this study, we highlight several implications and mitigation strategies that researchers and administrators can explore when encountering variability.

For System Administrators: Section VII-B shows that several NIC counters are strongly associated with performance variability. We expect that system administrators can modify resource managers to schedule workloads more intelligently or detect early signals of congestion [18]. For example, by periodically applying predictive methods similar to those proposed in this paper, administrators can proactively warn performance-sensitive users that they may be experiencing performance degradation. In fact, many supercomputers (e.g., Perlmutter) already collect Lightweight Distributed Metric Service (LDMS) [46] data in real time, which includes a rich set of network counters. Moreover, as demonstrated in Section VII-B, our model maintains high accuracy even for applications with limited training samples, demonstrating its generalizability. We believe this evidence is sufficient to support training a universal model using system-wide network counter data to predict ongoing performance degradation.

In addition, Section VI-B shows that an increased number of communication-heavy jobs running simultaneously can lead to slowdown for other jobs. Administrators can monitor and limit the number of such jobs run concurrently, or isolate them to a dedicated dragonfly group to keep the system healthy.

For Users: Our model demonstrates that performance variability predictions can be achieved even with limited training data. Users can leverage our approach by incorporating a small set of profiling data from their own applications. Whenever they are allocated compute nodes, they can predict variability at the start of their job. If significant performance degradation is detected, users can cancel the job early to save node hours.

IX. Conclusion

In this paper, we conducted a comprehensive investigation into performance variability on flagship GPU-based supercomputers, Perlmutter and Frontier. By executing a suite of representative HPC (AMG2023, MILC) and AI (DeepCAM, nanoGPT) applications over an extended period, we collected an extensive dataset including application runtime, detailed profiling data, NIC counters, and Slurm job logs. Our results reveal that although we observed inherent performance differences between individual GPUs, these differences did not significantly affect large-scale parallel workloads. Furthermore, our investigation revealed that job placement, specifically the number of dragonfly groups allocated to a job, did not significantly influence performance.

We found that performance degradation is not directly impacted by the total number of jobs or users running on the system but rather due to the concurrent jobs of a specific subset of "Top Users" running communication-intensive applications. By leveraging machine learning methods, we showed that network performance is the dominant driver of variability in GPU-based supercomputers. Across both HPC and AI workloads, run-to-run performance variability correlated strongly with network contention and communication delays. Moreover, our ML model accurately predicts runtime variability, even for applications with only a few samples in the training set.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 2047120. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy (DOE) Office of Science User Facility, operated under Contract No. DE-AC02-05CH11231 using NERSC awards DDR-ERCAP0034262 and ALCC-ERCAP0034775. This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] S. Singh, P. Singhania, A. Ranjan, J. Kirchenbauer, J. Geiping, Y. Wen, N. Jain, A. Hans, M. Shu, A. Tomar, T. Goldstein, and A. Bhatele, "Democratizing AI: Open-source scalable LLM training on GPU-based supercomputers," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '24, Nov. 2024.

[2] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on GPU clusters," *CoRR*, vol. abs/2104.04473, 2021.

[3] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, Y. Jia, S. He, H. Chen, Z. Bai, Q. Hou, S. Yan, D. Zhou, Y. Sheng, Z. Jiang, H. Xu, H. Wei, Z. Zhang, P. Nie, L. Zou, S. Zhao, L. Xiang, Z. Liu, Z. Li, X. Jia, J. Ye, X. Jin, and X. Liu, "MegaScale: Scaling large language model training to more than 10,000 GPUs," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 745–760. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[4] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," Tech. Rep., 2022.

[5] A. Bhatele, J. J. Thiagarajan, T. Groves, R. Anirudh, S. A. Smith, B. Cook, and D. K. Lowenthal, "The case of performance variability on dragonfly-based systems," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '20. IEEE Computer Society, May 2020.

[6] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503247

[7] S. Ramesh, M. Titov, M. Turilli, S. Jha, and A. Malony, "The ghost of performance reproducibility past," in *2022 IEEE 18th International Conference on e-Science (e-Science)*. IEEE, 2022, pp. 513–518.

[8] P. Sinha, A. Guliani, R. Jain, B. Tran, M. D. Sinclair, and S. Venkataraman, "Not all gpus are created equal: characterizing variability in large-scale, accelerator-rich systems," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 01–15.

[9] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*. IEEE Computer Society, 2008.

[10] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the slingshot interconnect," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[11] M. Salimi Beni, S. Hunold, and B. Cosenza, "Analysis and prediction of performance variability in large-scale computing systems," *The Journal of Supercomputing*, vol. 80, no. 10, pp. 14 978–15 005, 2024.

[12] Y. Zhang, T. Groves, B. Cook, N. J. Wright, and A. K. Coskun, "Quantifying the impact of network congestion on application performance and network metrics," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 162–168.

[13] X. Wang, M. Mubarak, X. Yang, R. B. Ross, and Z. Lan, "Trade-off study of localizing communication and balancing network traffic on a dragonfly system," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 1113–1122.

[14] M. Mubarak, P. Carns, J. Jenkins, J. Li, N. Jain, S. Snyder, R. B. Ross, C. D. Carothers, A. Bhatele, and K.-L. Ma, "Quantifying I/O and communication traffic interference on dragonfly networks equipped with burst buffers," in *Proceedings of the IEEE Cluster Conference*, ser. Cluster '17, Sep. 2017, lLNL-CONF-731482.

[15] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully! job interference study on dragonfly network," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 750–760.

[16] S. A. Smith, C. Cromey, D. K. Lowenthal, J. Domke, N. Jain, J. J. Thiagarajan, and A. Bhatele, "Mitigating inter-job interference using adaptive flow-aware routing," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18. IEEE Computer Society, Nov. 2018.

[17] S. A. Pollard, N. Jain, S. Herbein, and A. Bhatele, "Evaluation of an interference-free node allocation policy on fat-tree clusters," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '18. IEEE Computer Society, Nov. 2018.

[18] D. Nichols, A. Marathe, K. Shoga, T. Gamblin, and A. Bhatele, "Resource utilization aware job scheduling to mitigate performance variability," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '22. IEEE Computer Society, May 2022.

[19] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC'03)*, 2003.

[20] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on xeon phi based cray xc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–13.

[21] E. A. Leon, I. Karlin, A. Bhatele, S. H. Langer, C. Chambreau, L. H. Howell, T. D'Hooge, and M. L. Leininger, "Characterizing parallel scientific applications on commodity clusters: An empirical study of a tapered fat-tree," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Computer Society, Nov. 2016, lLNL-CONF-681011. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SC.2016.77

[22] K. Yoshida, R. Sakamoto, K. Sato, A. Bhatele, H. Yamaki, H. Honda, and S. Miwa, "VAHRM: Variation-aware resource management in heterogeneous supercomputing systems," *IEEE Transactions on Parallel & Distributed Systems*, vol. 36, no. 08, pp. 1713–1727, Aug. 2025. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TPDS.2025.3577252

[23] "Nccl," 2020. [Online]. Available: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html

[24] "Rccl," 2025. [Online]. Available: https://github.com/ROCm/rccl

[25] "Nersc-10 benchmark suite," 2024. [Online]. Available: https://www.nersc.gov/systems/nersc-10/benchmarks/

[26] "Olcf-6 benchmark suite," 2024. [Online]. Available: https://www.olcf.ornl.gov/draft-olcf-6-technical-requirements/benchmarks/

[27] R. Falgout, J. Jones, and U. Yang, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. Bruaset and A. Tveito, Eds. Springer-Verlag, 2006, vol. 51, pp. 267–294.

[28] R. Li and U. M. Yang, "Amg2023," [Computer Software] https://doi.org/10.11578/dc.20230413.1, mar 2023. [Online]. Available: https://doi.org/10.11578/dc.20230413.1

[29] C. Bernard, T. Burch, T. A. DeGrand, C. DeTar, S. Gottlieb, U. M. Heller, J. E. Hetrick, K. Orginos, B. Sugar, and D. Toussaint, "Scaling tests of the improved Kogut-Susskind quark action," *Physical Review D*, no. 61, 2000.

[30] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, "Solving lattice qcd systems of equations using mixed precision solvers on gpus," *Computer Physics Communications*, vol. 181, no. 9, pp. 1517–1528, 2010.

[31] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb, "Scaling lattice qcd beyond 100 gpus," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[32] Q. collaboration *et al.*, "Accelerating lattice qcd multigrid on gpus using fine-grained parallelization," *arXiv preprint arXiv:1612.07873*, 2016.

[33] R. Li, C. DeTar, S. Gottlieb, and D. Toussaint, "Milc code performance on high end cpu and gpu supercomputer clusters," 2017. [Online]. Available: https://arxiv.org/abs/1712.00143

[34] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, "Exascale deep learning for climate analytics," 2018. [Online]. Available: https://arxiv.org/abs/1810.01993

[35] S. Farrell, M. Emani, J. Balma, L. Drescher, A. Drozd, A. Fink, G. Fox, D. Kanter, T. Kurth, P. Mattson *et al.*, "Mlperf™ hpc: A holistic benchmark suite for scientific machine learning on hpc systems," in *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE, 2021, pp. 33–45.

[36] Prabhat, O. Rübel, S. Byna, K. Wu, F. Li, M. Wehner, and W. Bethel, "TECA: A Parallel Toolkit for Extreme Climate Analysis," *Procedia Computer Science*, vol. 9, pp. 866–876, 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050912002141

[37] S. Singh and A. Bhatele, "AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '22. IEEE Computer Society, May 2022.

[38] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[39] "Slurm workload manager," 2020. [Online]. Available: https://slurm.schedmd.com/documentation.html

[40] "Hpe cassini performance counters," 2024. [Online]. Available: https://cpe.ext.hpe.com/docs/latest/getting_started/HPE-Cassini-Performance-Counters.html

[41] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable mpi profiling," 2005.

[42] "Pytorch profiler," 2024. [Online]. Available: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

[43] "Holistic trace analysis (hta): A library to analyze pytorch traces," 2023. [Online]. Available: https://github.com/facebookresearch/HolisticTraceAnalysis

[44] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[45] D. Nichols, A. Movsesyan, J.-S. Yeom, D. Milroy, T. Patki, A. Sarkar, and A. Bhatele, "Predicting cross-architecture performance of parallel programs," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '24. IEEE Computer Society, May 2024.

[46] O. Cankur, B. Austin, D. Kulkarni, and A. Bhatele, "Characterizing production gpu workloads using system-wide telemetry data," 2025. [Online]. Available: https://arxiv.org/abs/2502.18680

Artifact Description (AD)

Appendix A
Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 We present a longitudinal performance-variability dataset on Perlmutter and Frontier across diverse workloads and system states.
- C_2 We analyze how hardware differences, job placement, concurrency, and network conditions affect application performance.
- C_3 We use machine learning to identify the metrics most predictive of performance variability for HPC and deep learning workloads.
- C_4 We show that network performance dominates GPU-cluster variability and provide actionable mitigation insights for architects and administrators.

B. Computational Artifacts

Artifact ID	Contributions Supported	Related Paper Elements
$A_1 - A_4$	C_1, C_2, C_3, C_4	Tables 1, 3 Figure 1, 2, 4, 7-13
A_5	C_1, C_2, C_3, C_4	Tables 3 Figures 6-7, 14
A_6	C_1, C_2, C_3, C_4	Tables 2, 3 Figures 1-14

- A_1 **AMG2023**: https://github.com/hpcgroup/AMG2023/tree/performance-variability
- A_2 **MILC**: https://github.com/hpcgroup/milc_qcd/tree/performance-variability
- A_3 **DeepCAM**
  Perlmutter: https://github.com/hpcgroup/hpc_results_v3.0/tree/performance-variability
  Frontier: https://github.com/hpcgroup/hpc/tree/performance-variability
- A_4 **nanoGPT**: https://github.com/axonn-ai/nanoGPT/tree/performance-variability
- A_5 **Micro_benchmarks**: https://github.com/hpcgroup/gpu-benchmarks/tree/performance-variability
- A_6 **Analyze Scripts**: https://github.com/hpcgroup/perf-variability

Appendix B
Artifact Identification

A. Computational Artifact $A_{1–4}$

We evaluate two HPC applications (AMG2023, MILC) and two AI training workloads (DeepCAM, nanoGPT), covering MPI and NCCL/RCCL communication patterns.

The output of computational artifacts A1-A4 will include detailed profiling logs and execution times. We use the detailed profiling breakdown, Cassini counter results, and job scheduling logs from these applications to analyze the sources of performance variability (A_6).

Expected Reproduction Time (in Minutes)

AMG2023 requires approximately 10 minutes of compilation time on both Frontier and Perlmutter, including the dependency library (hypre).

MILC requires approximately 15 minutes of compilation time, including the dependency library (QUDA).

nanoGPT and DeepCAM do not require compilation, but each requires Python environment setup (about 15 minutes). In addition, nanoGPT requires downloading GPT-2 weights (about 10 minutes). DeepCAM requires downloading an 8.8 TB dataset, which takes at least 60 minutes. On Perlmutter, the dataset must be converted to `.npy` format, which takes about 120 minutes. For DeepCAM on Perlmutter, we use the latest implementation from the MLPerf™ HPC Training v3.0 benchmark. However, this implementation cannot run on Frontier, so we use the v1.0 implementation on Frontier.

We configured the runtime for artifacts A1-A4 to be around five minutes each when running on 64 nodes. Considering initialization and log output overhead, the actual wall time for each run is approximately 10 minutes. The total number of runs and estimated execution time for the experiments presented in this paper are summarized below. The total estimated execution time across all jobs is 7610 minutes (approximately 127 hours), corresponding to approximately 8117 node hours.

TABLE IV
APPLICATIONS RUNS ON PERLMUTTER AND FRONTIER

Application	# of jobs	Estimated Time (minutes)	Node hours
Perlmutter			
AMG2023	104	1040	1109.3
MILC	78	780	832.0
DeepCAM	67	670	714.7
nanoGPT	94	940	1002.7
Perlmutter Total	**343**	**3430**	**3658.7**
Frontier			
AMG2023	168	1680	1792.0
MILC	38	380	405.3
DeepCAM	109	1090	1162.7
nanoGPT	103	1030	1100.0
Frontier Total	**418**	**4180**	**4460.0**
Grand Total	**761**	**7610**	**8118.7**

Artifact Setup (incl. Inputs)

Hardware: Perlmutter (GPU nodes) at NERSC and Frontier at OLCF.

Software: A1: hypre: https://github.com/hypre-space/hypre, mpiP

A2: QUDA: https://github.com/lattice/quda, mpiP

A3: Python, PyTorch, CUDA/ROCm

A4: AxoNN: https://github.com/axonn-ai/axonn, Python, PyTorch, CUDA/ROCm

Datasets / Inputs: AMG2023 and MILC do not require any additional datasets.

DeepCAM requires downloading the 8.8TB Full dataset. Dataset instructions are available at https://gitlab.com/NERSC/N10-benchmarks/deepcam/-/blob/main/data/globus.md.

nanoGPT requires downloading the GPT2 dataset (OpenWebText2). Information can be found at https://openwebtext2.readthedocs.io/en/latest/. We provide a script at https://github.com/axonn-ai/nanoGPT/blob/master/data/openwebtext/prepare.py

Installation and Deployment: Detailed installation and deployment instructions for each application (A1-A4) and the micro-benchmarks (A5) are provided within the documentation of the Analyze_scripts artifact (A6). This includes specific compiler versions, environment setup, and execution commands for both Perlmutter and Frontier systems.

Artifact Execution

We use scrontab/crontab to periodically submit scripts that run the four applications (A1-A4). We record the execution time and JobID for each run.

Artifact Analysis (incl. Outputs)

The execution of artifacts A1-A4 will generate application execution logs, Cassini counter logs, and profiling logs (e.g., from mpiP or PyTorch profiler). We use the analysis tools provided in artifact A6 to process and analyze these logs to understand performance characteristics and variability. The workflow is essentially: Run A1/A2/A3/A4 $\rightarrow$ Generate Logs $\rightarrow$ Use A6 to Analyze Logs.

B. Computational Artifact A_5

This artifact contains micro-benchmarks including NCCL/RCCL allreduce, MPI allreduce, and FP16 GEMM tests. We incorporate the execution of these micro-benchmarks into the `srun` command before running the main applications (A1-A4).

Expected Reproduction Time (in Minutes)

The allreduce and GEMM tests take approximately 5 minutes in total per execution. Since these benchmarks are run preceding each job of A1-A4, the total execution time attributed to A5 corresponds to the total number of application jobs. Total runs: 761 (343 on Perlmutter, 418 on Frontier). Total estimated time: Approximately 3805 minutes (around 63.4 hours). Total node hours: Approximately 4185.3 node hours (3805 min / 60 min/hr * 64 nodes).

Artifact Setup (incl. Inputs)

Software: cuBLAS/rocBLAS, RCCL/NCCL, Cray MPICH
Datasets / Inputs: GEMM: input size 32768x32768x32768
MPI Allreduce: 1 KB, 2 KB, ..., 1 MB
RCCL/NCCL Allreduce: 16 MB, 32 MB, ..., 2 GB

Artifact Execution

The workflow involves running the micro-benchmarks just before each run of the main applications (A1-A4). Workflow: Run A5 $\rightarrow$ Run A1/A2/A3/A4 $\rightarrow$ correlate micro-benchmarks results with applications execution time.

Artifact Analysis (incl. Outputs)

The output consists of performance measurements (e.g., bandwidth for allreduce, execution time for GEMM) for the specific node allocation at the time of execution.

C. Computational Artifact A_6

This artifact (A6) contains the analysis scripts used to process the data generated by artifacts A1-A5. It extracts information from the logs produced by A1-A5 to analyze overall performance variability (Fig. 1), detailed routine breakdowns (Fig. 2-5), GEMM variability (Fig. 6-7), and the impact of various factors on performance variability (Fig. 8-14). Additionally, A6 includes the job submission scripts used for the experiments and detailed build/installation scripts for artifacts A1-A5.

Expected Reproduction Time (in Minutes)

All analysis scripts in A6 are written in Python and do not require compilation. Running all analysis scripts takes 30 minutes on a login node of Perlmutter or Frontier.

Artifact Setup (incl. Inputs)

Hardware: Login nodes of Perlmutter at NERSC and Frontier at OLCF.

Software: Python, PyTorch, HTA (Holistic Trace Analysis). HTA for Frontier: https://github.com/hpcgroup/perf-variability

Datasets / Inputs: The output logs generated by artifacts A1-A5 and the Slurm logs from Perlmutter and Frontier. The total raw data amounts to 10TB for each system.

Artifact Execution

The experimental workflow involves running the Python scripts provided in A6. These scripts parse the log files from A1-A5 runs and Slurm logs, perform data processing and analysis, and generate the data points used for the figures in the paper. Workflow: Collect Logs (from A1-A5, Slurm) $\rightarrow$ Run A6 Python Scripts $\rightarrow$ Generate Figure Data.

Artifact Analysis (incl. Outputs)

The primary output of A6 is the raw data used to generate all figures (Figures 1-14) presented in the paper. This data is typically saved in formats such as CSV. Then we plot figures with the data.