VaBO: Delivering Reliable Autotuning Performance in Parallel Applications under Unreliable Conditions

Vedica Rao
Santa Clara University
Santa Clara, California, USA
vrao3@scu.edu

Saeyeon Kim*
Santa Clara University
Santa Clara, California, USA
saeyeonkim026@gmail.com

Hengrui Luo
Rice University
Houston, Texas, USA
hl180@rice.edu

Tirthak Patel
Rice University
Houston, Texas, USA
tp53@rice.edu

Younghyun Cho†
Santa Clara University
Santa Clara, California, USA
younghyun.cho@scu.edu

Abstract

Empirical autotuning has emerged as an essential tool for the performance engineering of parallel and High-Performance Computing (HPC) applications. In this context, black-box optimization techniques, such as Bayesian optimization (BO), are widely adopted to guide the search for optimal configurations within a limited number of true performance samples. While existing work has primarily focused on improving the sample efficiency of autotuning methodologies, computer systems often exhibit runtime performance variability stemming from diverse sources, such as compute noise, memory noise, I/O interference, and OS jitter. Such variability can mislead autotuners, causing them to select sub-optimal configurations or waste evaluations.

This paper presents VaBO, Variability-Aware BO-based Tuning, a technique that identifies unstable sample evaluations during tuning and re-evaluates them before they are used in the optimization process. Our technique employs a lightweight performance-variability detection unit that requires no modifications to the target tuning code. When a target program is evaluated for a given parameter configuration, our variability detection unit measures the performance variance within the run with low-overhead library call interceptions. Then, the variability-aware tuning algorithm estimates sample reliability during BO iterations using dynamic thresholding and clustering of the performance statistics. We evaluate VaBO on several HPC applications, including Neural Network Inference, ScaLAPACK PDGEQRF, SuperLU_DIST, and Hypre. From our experiments, under runtime variability, overall, VaBO attains about 4-8% better tuned result for real-world HPC codes such as ScaLAPACK's PDGEQRF and SuperLU_DIST, compared to an existing HPC autotuner, GPTune.

CCS Concepts

• **General and reference** → **Performance**; • **Computing methodologies** → *Parallel computing methodologies.*

Keywords

Autotuning, performance variability, Bayesian optimization

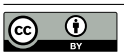
ACM Reference Format:
Vedica Rao, Saeyeon Kim, Hengrui Luo, Tirthak Patel, and Younghyun Cho. 2026. VaBO: Delivering Reliable Autotuning Performance in Parallel Applications under Unreliable Conditions. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3838684.3846861

1 Introduction

Modern parallel and high-performance computing (HPC) codes typically exhibit an increasing number of tuning parameters for algorithmic choices (e.g., tiling/blocking parameters for math libraries [1, 15, 16], sparsity formats [32, 48]) as well as runtime parameters for parallel programming models [12, 21, 24, 26]. Historically, analytical performance modeling has been used to understand, model, and optimize code performance [30, 51]. Such an approach is no longer feasible as hardware systems continue to evolve and tuning problems involve an increasing number of tunable parameters. Alternatively, machine learning (ML)-based optimizations, that apply trained ML models to optimize code performance, have also recently been applied [17–19, 23]. However they require significant effort to collect and train high-quality training data, which often limits their practical applicability.

To address these challenges, *empirical automated performance tuning (autotuning)* has become a popular choice for optimizing tunable parameters in high performance and performance critical codes [3, 9, 14, 33, 35, 47, 52], serving as a sustainable tool for performance engineering in the current exascale era. Unlike methods relying on deep human expertise (e.g., analytical modeling) or extensive prior data collection (e.g., pre-trained ML prediction model), empirical autotuning typically employs a data-driven (black-box) optimization to find the best tuning parameters for a given prescribed tuning budget. In particular, Bayesian optimization (BO) based approaches are gaining significant attention. The approach,

*Now with AMD; work done while at Santa Clara University
†Corresponding author



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846861

which relies on true code function evaluations, aims to identify the best parameter configurations within a limited tuning budget.

In the domain of application tuning, existing research has predominantly focused on improving the sample efficiency of BO methodologies, such as through advanced surrogate modeling and multitask/transfer learning, to achieve high-quality tuning results with fewer function evaluations [9, 33, 35, 36, 39, 40]. In this work, however, we address another significant challenge in deploying empirical tuning at scale in modern performance critical computing infrastructure: at-runtime performance variability that degrades the quality of samples. When randomness or variability of samples exists, the conventional rule of thumb has been to conduct repeated evaluations for a sample to ensure reliability, which renders the tuning process prohibitively expensive. However, ignoring sample quality can also lead to a significant drop in the accuracy of the surrogate model when the reliability of collected samples is low.

In this work, we shift the paradigm of BO tuning research from sample efficiency to reliability of performance samples, i.e., the reliability of function evaluations of HPC codes. Such applications often suffer from run to run performance variability, particularly on large-scale and shared-user systems. Such runtime performance variances can occur due to many factors, such as interventions from other noisy workloads [27, 38, 45] and non-deterministic turnaround times of operations to input/output (I/O) devices [4, 11, 34]. Instead of offloading the burden of ensuring the quality of samples to the user, our key motivation is to utilize a light-weight variability detection technique to identify the quality of the obtained sample during each function evaluation of the BO process, however, this must be done without incurring a significant runtime overhead that would distort the objective value of the sample itself.

To this end, we present VaBO, which is a Performance **V**ariability-**A**ware **BO**-based autotuner for application performance tuning. Unlike existing (BO-based) autotuners [3, 9, 14, 33, 35, 47, 50, 52], we integrate a light-weight performance variability detection unit directly when a function evaluation is being done from the BO loop. We build VaBO by extending an existing open-source BO-based HPC tuner called GPTune [8], incorporating our in-house variability detection unit and a variability-aware tuning algorithm. While various techniques exist for variability analysis and detection [43–45, 53], one of our critical design considerations for VaBO is minimizing runtime overhead to enable real-time sample quality assessment during function evaluation while not invalidating the function evaluation result. Our performance variability detector monitors the performance trends of the repeated kernels within a given target application (e.g., GEMM kernels [16]) to identify the presence of system noise during the function evaluation. Consequently, our variability-aware tuning algorithm adopts a simple re-run strategy; if the reliability of a sample is below a threshold, we give a second chance to evaluate the same parameter configuration.

We evaluate VaBO with several application tuning scenarios under controlled variability conditions created by various synthetic noise generators. We first demonstrate the efficacy of VaBO using a Neural Network inference workload tuning task. Then, we apply VaBO for several critical, real-world HPC application tuning scenarios, including tuning for ScaLAPACK's PDGEQRF [7], SuperLU_DIST's LU factorizations [32], and Hypre [20]. From our evaluations, VaBO consistently achieves higher tuning quality than the baseline autotuning method such as GPTune [33] that does not account for variability detection. We also report our detection accuracy and its runtime overhead, showing that VaBO's performance variability detection can be used during BO function evaluations.

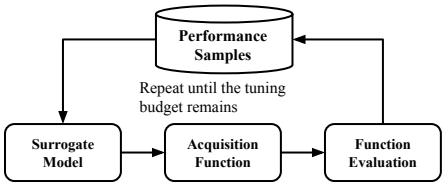


Figure 1: Basic procedure of Bayesian optimization.

In summary, our main contributions are:

- A lightweight performance variability detection unit, enabling a reliability check for BO's function evaluation, without incurring noticeable runtime overhead.
- A variability-aware tuning algorithm that performs an adaptive re-evaluation strategy based on the lightweight variability detection unit.
- A comprehensive tuning framework VaBO, that inherits the various features of the open-source BO tuner GPTune, while extending its capability to become a variability-aware tuner.
- Evaluations of VaBO with real-world HPC tuning problems such as ScaLAPACK, SuperLU_DIST, and Hypre.

The remainder of this paper is organized as follows. Section 2 provides background knowledge on BO and performance variability detection. In Section 3, we present the detailed techniques of VaBO. We present our evaluation results in Section 4. We discuss related work in Section 5 and conclude this paper in Section 6.

2 Background

Here, we describe the concepts and techniques related to this work.

2.1 Bayesian Optimization

Bayesian Optimization (BO) [22] is a popular empirical autotuning technique used for optimization of large, expensive-to-evaluate objective functions. The approach uses statistical models to identify optimal configurations without exhaustively exploring the entire search space. As shown in Figure 1, in the first step, a surrogate model, such as a Gaussian Process (GP) regression model [25], is created using a set of initial, random samples, which provides both a predicted value and an uncertainty quantification for unseen samples. Then, an acquisition function, such as Expected Improvement (EI), selects the most promising next candidate by balancing exploration and exploitation. Third, the objective function is evaluated using the selected configuration, and the resulting sample is used to update the surrogate model. This iterative process continues until convergence is met or the tuning budget is exhausted.

Inconsistent or unreliable evaluation results can potentially mislead and divert the surrogate modeling process from the true optimal configurations. Our work advocates the use of BO, aligning with the recent trends in autotuning. However, unlike existing techniques, we distinguish our work by focusing on the reliability of samples. To our best knowledge, this area remains underexplored.

2.2 Runtime Performance Variability Detection

Performance variability in computing systems can arise from numerous sources, including nonuniform compute resource allocation, varying memory access and I/O latencies with resource contention. Such variability makes it difficult to achieve consistent performance across program runs, which is particularly important in BO autotuning. Traditional techniques for identifying performance variability typically use repeated runs, detailed tracing, or online benchmarking of the application. While effective, these approaches are inherently intrusive or add significant runtime overhead to the performance. Furthermore, re-running applications multiple times is typically impractical for large-scale applications.

To overcome these limitations, a lightweight detection technique is required. A method capable of detecting the variability of the program in a single run, and with no changes to the program code (or minimal changes). Several existing works, such as vSensor [45] and Vapro [53], utilized fixed-size workloads, i.e., invocations of specific code functions with identical or similar workload volumes.

Why we adopt workload snippet-based analysis for variability detection? Here, a snippet refers to an identical code function with similar workload volumes. In other words, our analysis is performed on selected workload snippets, as opposed to heavy profiling or online benchmarking of the entire program. We consider such statistical analysis of workload performance to be a promising approach for detecting performance variance during BO function evaluations. First, the code patterns of many HPC and AI workloads are iterative in nature, involving the repeated invocation of computation kernels and function calls. When system noise or interference occurs during an application run, the statistical variance of these repeated kernel execution times can serve as an indicator of external noise.

However, in applying this method to BO tuning, a significant challenge lies in the overhead of variability analysis. Existing methods [53], typically designed as post-run analysis and profiling tools, intercept and profile a vast number of function calls. For instance, Vapro intercepts and analyzes the majority of MPI routines (e.g., MPI_Send, MPI_Receive, etc.) to construct a comprehensive performance profile. In the context of BO, such heavy instrumentation can incur noticeable overhead, which may adversely affect the fidelity of the performance sample itself. Note that in empirical autotuning, the measured execution time is the objective function value itself. Therefore, any overhead introduced by the variability detector directly distorts the tuning metric. Consequently, a significantly more lightweight detection is required for VaBO.

In VaBO, we aim to balance between the precision of variability analysis and the overhead of the detection. To do so, we adopt a user-guided approach, where the user can designate a specific routine/function to serve as the variance identifier kernel. This approach remains practical, as HPC application tuning is typically conducted by the domain experts or users who possess a reasonable understanding of the target application's critical kernels.

3 VaBO

In this section, we discuss the details of VaBO, a runtime Performance Variability-aware Bayesian Optimization autotuner.

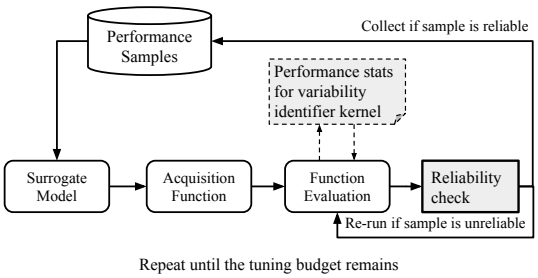


Figure 2: VaBO Overview.

3.1 Overview

Figure 2 illustrates the overview of our framework. The VaBO workflow has the following stages:

(1) **Baseline calibration**: The first, initial samples for surrogate modelling in BO are also leveraged to establish an adaptive *baseline variability threshold* (V_{thresh}). This threshold represents the expected noise level of the system hardware.
(2) **Iterative parameter exploration**: In each BO tuning iteration, a tuning parameter sample configuration is selected. This configuration is executed with the *variability detection module*, which monitors and logs performance data for the variability identifier kernel.
(3) **Variability score computation and re-evaluation policy**: The interception tool provides a variability score for each evaluation and triggers an automatic re-evaluation of the configuration for unstable samples.

In what follows, we detail how we detect runtime performance variability and then discuss the algorithm for re-evaluation policy.

3.2 Runtime Variability Detector

Our approach builds on the observation that many HPC applications execute the same or similar code regions and/or kernels many times. For example, applications such as neural networks repeatedly invoke the same mathematical kernels (e.g., GEMM) that perform the same computation over different input batches. Since the workload computation pattern remains relatively constant or predictable, the fixed-workload code snippets are expected to exhibit consistent execution time across the iterations, as long as they execute for the same input sizes. Therefore, deviations from this expected behavior can be used as indicators of performance variability.

This principle has been previously demonstrated in the VAPRO framework [53], which used repeated execution code regions to detect intra-run performance variation. Our technique adapts and extends this approach for variability-aware tuning. However, unlike VAPRO, which examines various code regions, we intercept only a few designated, important indicator kernels in order to minimize the runtime overhead. In the context of variability-aware BO tuning, as the obtained performance record is used as the performance of the code, the lightweight nature of the variability detection tool is more critical than the absolute accuracy to maintain the efficiency of the autotuning process.

Our approach captures the performance variability with a mechanism that (1) selects variability identifier kernels, (2) dynamically

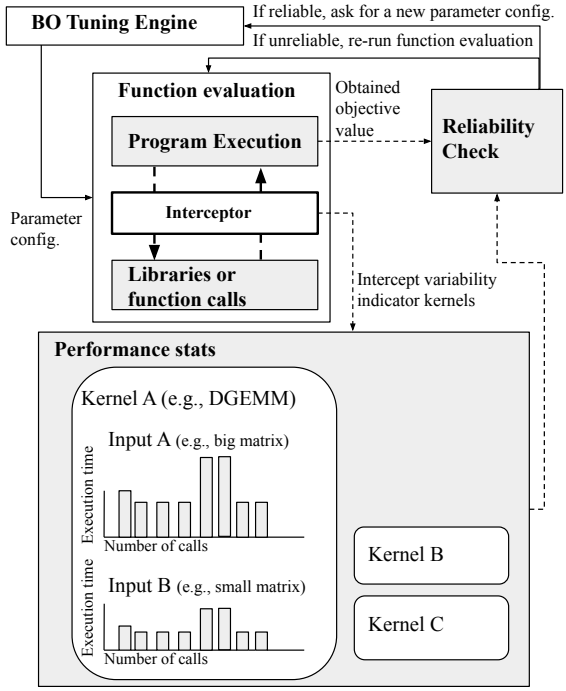


Figure 3: The Workflow of our Variability Detection Module.

intercepts the kernels during runtime, (3) records their execution times and performance metrics, (4) performs workload clustering (5) computes a variability score aggregating per-workload variances.

3.2.1 Kernel Selection. Many parallel and HPC applications repeatedly execute kernels whose workload size can be determined directly from the function arguments. These operations typically have a consistent number of floating-point operations (FLOPs) or memory operations given fixed inputs. Therefore, deviations in their execution time can be attributed to external runtime noise and not internal variances such as branching or memory access pattern changes.

Common examples include BLAS [16] routines such as GEMM and GEMV, or common routines in a parallel programming model (e.g., MPI_Send/Receive). These predictable workload kernels can serve as variability identifier kernels. In this work, the variability identifier kernels are specified by the user.

3.2.2 Runtime Interception. During the execution of the application sample, every call of the indicator kernel is dynamically intercepted using `LD_PRELOAD`. We implement the interception using a lightweight wrapper library that is loaded at runtime. The wrapper defines an interception function for the selected variability identifier kernel and is invoked when the corresponding kernel is called. Upon interception, the wrapper invokes the original kernel, and measures its execution time, along with metadata used to verify workload-size consistency, including relevant function arguments and hardware performance counters. For each kernel invocation, this information is recorded, and then control is returned to the application. This approach allows performance collection of the kernels without application source code modification.

Each intercepted call corresponds to a kernel that performs a known quantity of computation and is associated with a *workload proxy* W. This proxy is a static metric that is defined to capture the computational or communication complexity of the kernel. For computation kernels, for example, W is typically defined by the total floating-point operations. For communication kernels, on the other hand, W can be defined as the total message size in bytes. In cases where the analytical complexity is difficult to derive, hardware performance counters (such as total instructions from PAPI [28]) can serve as an empirical workload proxy.

3.2.3 Workload Grouping. Directly comparing execution times across all kernel instances is not meaningful, since runtimes naturally increase with workload size. To enable fair performance comparisons, we group kernel instances into fixed workload-size segments using static binning.

A logarithmic scale is used because workloads with larger problem sizes can span several orders of magnitude, and performance differences typically scale proportionally rather than linearly.

Each kernel instance with workload W is assigned to a bin index j defined by:

$$j = \lfloor \ln(W)/\Delta \rfloor$$

where Δ is the bin width set by default as 0.1.

The bin width was empirically selected by performing a parameter sensitivity sweep on execution logs. This value, found to be 0.1, represents the optimal "elbow point" that balances two metrics, data retention and inter-bin variability. Data retention here is defined as the percentage of samples that fall in bins with at least 5 samples; bins with fewer than 5 samples were excluded as they do not contain enough observations for meaningful analysis. The intra-bin variability metric measures the standard deviation of the runtimes within each bin.

For each tested value of Δ, kernel instances were grouped into logarithmic workload bins and evaluated using the two metrics. Very small values of Δ created many fine-grained bins with few samples. We observed that setting $\Delta = 0.01$ caused 35% of the data to be discarded. At $\Delta = 0.1$, we found data retention at $> 95\%$. Larger values of Δ led to grouping together kernel instances with different workloads, which increased the variance within bins. Based on these results, in our experiment setting, $\Delta = 0.1$ was selected because it retains most of the samples while ensuring workloads within each bin are sufficiently similar for meaningful performance comparison.

3.3 Variability Score Calculation

In VaBO, when tuning an MPI application, each MPI process collects the performance data of the variability identifier kernel instances executed by the process. For calculating variability score of an application execution, after the execution of an evaluation sample, for each bin j, the interception library aggregates profiling values (count n_j, execution time sum $\sum_i t_{i,j}$, and sum of squares $\sum_i t_{i,j}^2$) across all MPI processes using `MPI_Reduce`, where $t_{i,j}$ denotes the runtime of the i-th invocation in bin j, and n_j represents the total number of interceptions. Similarly, when tuning an OpenMP application, each thread collects the profiling data from each thread, and we aggregate them for the variability score calculation.

For each valid bin j with sufficient samples, we compute the standard deviation σ_j of the execution times as:

$$\sigma_j = \sqrt{\frac{\sum t_{i,j}^2}{n_j} - \left(\frac{\sum t_{i,j}}{n_j}\right)^2}$$

The aggregated weighted standard deviation σ_{agg} is calculated as a weighted sum of the standard deviations across all valid bins:

$$\sigma_{agg} = \sum_j \sigma_j \cdot w_j = \sum_j \sigma_j \cdot (j \cdot \Delta)$$

The weighting factor $w_j = j \cdot \Delta \approx \ln(W_j)$ ensures that variability in larger, computationally intensive workloads contribute more significantly to the overall score than variability in smaller workloads.

To compute the final variability score V, in $[0, 1]$, we apply a hyperbolic tangent function to the aggregated weighted standard deviation. The final normalized variability score V is computed as:

$$V = \tanh\left(\frac{\sigma_{agg}}{\alpha}\right) \tag{1}$$

where α is a scaling constant that controls how sensitive the score is to changes in variability. For our setting, α is set to a default of 0.01 seconds (10 ms) chosen based on the observed variability in the tested HPC applications under no injected noise scenario. This ensures that typical system noise produces a low score close to zero while larger deviations result in a score approaching 1.

The tanh function is useful because it compresses the values into a bounded range [0-1], suppressing minor fluctuations while highlighting larger deviations.

A high final variability score V indicates that the observed runtime of the tuning sample is highly sensitive to system noise. This would trigger the VaBO tuner to re-evaluate the configuration.

3.4 Tuning Algorithm

Algorithm 1 describes the tuning algorithm with the variability detection unit. Our algorithm utilizes a dynamic thresholding mechanism. The process begins by collecting a set of I initial samples, which are used both to construct the surrogate model and to establish a baseline for the variability detection module. The number of initial samples is set to a default value of 5, but can be configured by the user. As the tuning loop runs, the variability scores of all previously evaluated configurations are partitioned into two clusters using K-means algorithm. The threshold level, V_{thresh}, is then dynamically calculated as the midpoint between the centers of the stable and unstable clusters. This way, rather than requiring a manually tuned threshold, the boundary between stable and unstable behavior is inferred directly from the observed data.

To verify how VaBO would perform in a noise-free environment, we incorporate a forced clustering check: the threshold is only applied if the distance between cluster centers exceeds $\delta_{min} = 0.15$. In a no-noise environment, all variability scores remain near zero; this check prevents the algorithm from force partitioning clean runs into two clusters and triggering unnecessary re-executions.

When a configuration is identified as unstable, meaning its variability score exceeds V_thresh, the tuner triggers a re-evaluation of the same configuration before passing the result to the surrogate model. This helps filter out any transient noise and ensures that the surrogate model is trained on reliable measurements. Since the clustering is recomputed as new evaluations are added, the threshold naturally evolves over time, making the method also robust to shifts in system behavior during the tuning process. The computational cost of this step is also negligible, as K-means with two clusters is inexpensive even when run repeatedly throughout the loop.

Algorithm 1 Variability-aware Tuning Algorithm.

```
Collect initial samples I
Initialize surrogate model with initial samples I
Initialize variability history H_V with initial samples I
while tuning budget not exhausted do
    Select next configuration c using Acquisition Function
    Execute configuration c
    Measure runtime T(c) and variability score V(c) (Equation 1)
    H_V ← H_V ∪ {V(c)}
    {C_low, C_high} ← K-Means(H_V, K = 2)
    V_thresh ← midpoint(center(C_low), center(C_high))
    if V(c) > V_thresh then
        Unstable sample detected; re-execute configuration c
        Update T(c), V(c) with new measurement
        H_V ← H_V ∪ {V(c)}
    end if
    Update surrogate model with sample (c, T(c))
end while
Return optimal configuration with minimum predicted runtime
```

4 Evaluation

We evaluate the effectiveness of VaBO with several case studies on HPC applications, using in-house Neural Network inference workloads, ScaLAPACK PDGEQRF [7], SuperLU_DIST [32], and Hypre [20].

4.1 Introducing Synthetic Noise

HPC applications can experience runtime performance variability due to various types of runtime interference or system jitter. In order to assess the potential of VaBO's techniques, we simulate synthetic noise during the tuning process. We mainly use two distinct types of noise generation schemes.

- **Compute contention:** To simulate CPU-bound interference, we execute a background process that performs intensive arithmetic operations, by executing large integer multiplications or floating-point matrix multiplications in parallel with the function execution of the tuning. The multiplication operations are compute-intensive and stress the CPU cores, leading to increased contention for the processor resources.
- **Memory contention:** To simulate memory hierarchy pressure, we execute STREAM-benchmark [37] like memory-intensive operations. Memory operations in the STREAM benchmark are designed to stress the memory bandwidth by repeatedly reading and writing large arrays of data. We implement the STREAM Triad kernel ($a_i = b_i + s \cdot c_i$), involving 2 memory reads and 1 memory write per element. We allocate 3 arrays of (1024 * 1024 * 64) doubles which is significantly larger than typical L3 cache. This is to ensure memory access and contention.

Injection Patterns. The execution pattern of the noise consists of randomized periods of bursts and sleep. The burst period runs the compute/ memory intensive loop for a random duration between 1ms and 10ms. The sleep period sleeps for a random duration between 0ms and 0.5ms after each burst. This forces context switching creating 'jitter' rather than just a slowdown.

For the applications SuperLU_DIST, ScaLAPACK and Hypre, the Slurm allocation uses `ntasks` equal to the maximum number of MPI processes required by any tuning configuration, with `cpus-per-task=1`. For the neural network inference tuning experiment in Section 4.3, one task is allocated with the maximum number of threads (48) specified by cpu-per-task. Noise processes are launched within the same allocation using mpirun –oversubscribe and are assigned a configurable fraction of the available CPU resources. The main experiments use all allocated cores for noise injection, while sensitivity experiments in Section 4.7.1 vary the fraction to evaluate the effect of contention intensity.

For experiments in sections 4.3 to 4.6 and 4.9 the noise is injected deterministically every Nth sample run. This deterministic approach enables experimental control and reproducibility, but also reflects patterns that occur in real-world systems since singular system interference is typically structured and periodic rather than random in multi-tenant servers. The experiments in Section 4.10 explore other noise patterns.

4.2 Tuning Scenarios

We evaluate how VaBO's technique performs under different tuning conditions:

- **BO with no runtime variability.** This scenario represents the baseline in ideal situations, where the standard BO tuner operates in a noise-free environment. This serves as the upper bound for tuning convergence speed. Our tuning setup is based on an open-source BO tuner, GPTune [33].
- **BO with runtime variability.** This scenario represents the current situation in a realistic, noisy environment. The standard BO tuner operates without any variability awareness.
- **VaBO with runtime variability (Our Proposed Technique).** This demonstrates VaBO with its variability detection unit and variability-aware algorithm. Once it determines an unreliable sample, it conducts a re-run in order to collect a reliable sample.
- **VaBO with no runtime variability** This demonstrates VaBO with its variability detection unit and variability-aware algorithm but in a noise-free environment. This serves to show that the overhead of VaBO does not significantly impact the tuning result.
- **VaBO with oracle runtime variability (`VaBO-Perfect`).** This demonstrates VaBO, but here we apply a clairvoyant setting, assuming it magically, perfectly knows when variance happens. This serves as an upper bound of VaBO when the variability detection unit works perfectly.

As an alternative baseline, one could adopt **"BO with multiple repeats for every sample"**, e.g., 3-5 multiple evaluations for every sample, regardless of whether the noise is present or not, to statically smooth out noise. However, from our observations, this configuration simply leads to prohibitively large tuning costs, as it performs multiple runs for each configuration. Or, it can be calculated as effectively multiplying the tuning budget by the repetition factor. Thus, for baselines, we mainly consider **"BO with no runtime variability"** for upper-bound performance and **"BO with runtime variability"** for comparison.

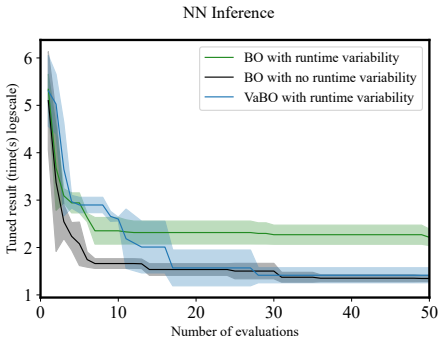


Figure 4: Tuning performance for Neural Network (NN) inference time. Mean of 3 runs; shaded area denotes standard deviation.

Essentially, VaBO is a layer over sample evaluation for reliability; it is orthogonal to the specific BO tuners. Varying the BO tuners would confound acquisition-level differences. Our with/without comparison on an identical tuner (GPTune) isolates that effect, allowing us to evaluate the efficacy of VaBO clearly.

Evaluation Metric. We compare the tuning quality of these tuners based on the *"tuned results against the number of evaluations"*, i.e., the best-found objective value as a function of the cumulative number of sample evaluations. Crucially, the number of evaluations for VaBO includes all the re-evaluations incurred by our re-run strategy, ensuring a fair comparison of total tuning cost.

4.3 Neural Network Inferences

We begin our evaluation with a Neural Network (NN) inference workload, a natural setting to explore runtime variability due to their repeated linear operations performed across many layers. This repetitive structure makes deviations in execution time easy to observe and quantify.

Experimental setup. The NN model and the environment variability are synthetically created to provide a controlled setting to observe the runtime fluctuations and assess the effectiveness of variability-aware tuning. We design a small feed-forward synthetic neural network where every linear layer is replaced with a custom implementation that records the execution time for each call. The model consists of 10 linear layers with a hidden dimension of 2048. The hidden layers are set with identical dimensions to ensure every call performs the same computation workload. Since these layers are executed repeatedly during inference across a test set of $2,000$ input vectors (from a total of $10,000$ synthetic vectors), they give a good indication of runtime variability.

The inference workload was parallelized using OpenMP on a dual-socket Intel Xeon Platinum 8168 processors (2x24 physical cores). The tuning search space consists of two tunable parameters, the number of threads and the matrix multiply tiling size.

- The matrix multiply tiling size: explored in range $[16, 512]$ with a step size of 16, i.e., $\{16, 32, 48, 64, \ldots, 512\}$.

- The number of OpenMP threads: For the number of threads was set between 8 and 48 in increments of 8, {8, 16, 24, 32, 40, 48}.

Experiments and results. Figure 4 reports the tuning results, comparing VaBO with the baseline tuning settings. For each linear layer interception, we record the runtime and compute the variability simply by the coefficient of variation (CV). Based on initial calibration phase in a quiescent environment (which is separate from the tuning results compared in Figure 4), we obtained a baseline CV to be approximately 0.6. This serves as the expected variability threshold under normal runtime conditions. When tuning, any sample run that exceeds this threshold is classified as high variability and is rerun to get a stable sample. In an environment that induces runtime performance variability, the sample runs experience unstable measurements, consequently resulting in slower convergence. For the noisy environment, we run a background stressor script of irregular large integer multiplication calculations to simulate compute noise and compare the tuning performance with and without the variability-aware tuning method. The noise was added every 3rd run with compute noise, one noise process for every thread (48 threads).

From these results, we observe that VaBO under runtime variability consistently finds a significantly better optimal tuning configuration than **BO with runtime variability** with the same tuning budget. Our VaBO achieves almost the same final tuned result with the noise-free setting, **BO with no runtime variability**, which indicates almost ideal result from variability-aware tuning. On the other hand, the regular BO tuner, without detecting noisy samples, leads to a significantly lower quality tuning, still not converging until 50 evaluations.

It is worth noting that this NN inference setting makes detecting runtime variability relatively straightforward due to the highly repetitive nature of the workload and the absence of other complex behaviors in the workload. Consequently, VaBO (**VaBO with runtime variability**) provides superior performance than the baseline. In the following subsections, for real-world HPC applications, we also conduct more detailed analysis including comparison of VaBO against the theoretical upper bound, **VaBO-Perfect with oracle runtime variability.**

4.4 ScaLAPACK PDGEQRF

Here, we use the tuning problem of ScaLAPACK's parallel QR factorization, PDGEQRF [7].

Experimental setup. The objective of the tuning problem is to find the optimal configurations that minimize the execution time of the PDGEQRF routine. For the search space, we consider the following four parameters:

- **mb:** Integer ranging from [1, 16]. This is a row block size multiplier. Here, $mb * 8$ represents the number of row block size.
- **nb:** Integer ranging from [1, 16]. This is a column block size multiplier, and $nb * 8$ is the column block size.
- **lg2npernode:** Integer ranging from [0, 4]. The number of processes per node is calculated as $2^{log2npernode}$.
- **p:** Integer ranging from [1, 16]. Here, p defines the number of row processes (the number of column processes is calculated based on the number of total processes and the number of row processes).

Our experiments were conducted on a dual-socket multicore system, 2x AMD EPYC 9654 processors, thus featuring NUMA properties. The machine has a total of 192 (2x96) physical cores and 2 hyperthreads per core (thus, a total of 384 logical cores). The ScaLAPACK code was executed using OpenMPI version 4.1.5 for an input matrix size of $m = n = 10,000$. To introduce runtime variability, we injected memory noise using a synthetic stressor script running STREAM triad-like memory operations. This interference was introduced deterministically every fourth run.

Experiments and results. For variability identifier kernel, we selected the MPI_Send routine as the variability identifier kernel, along with the message size as the workload proxy, based on the rationale that the user may consider an MPI routine would be an important performance indicator in a distributed-memory parallel algorithm in libraries such as ScaLAPACK.

Figure 5 shows tuning results for the abovementioned experimental setup. VaBO (**VaBO with runtime variability**) demonstrates robust tuning quality, achieving a higher tuning quality than the standard baseline. With 50 evaluations, VaBO achieves a final objective runtime value of 2.92s, approximately an 8.5% improvement over the 3.19s obtained by standard BO under the same noisy conditions.

4.5 SuperLU_DIST

SuperLU_DIST is a distributed-memory sparse direct solver. Here, we tune the 2D version (2D block-cyclic process layout) of SuperLU_DIST on a multi-socket multicore CPU node.

Experimental setup. This tuning problem has six tuning parameters as follows:

- **COLPERM:** Categorical option of {'2', '4'}; column permutation option.
- **LOOKAHEAD:** Integer range of [5, 20]
- **NSUP:** Integer range of [30, 1000]; maximum supernode size
- **NREL:** Integer range of [10, 60]; supernode relaxation parameter
- **nprows:** Integer range of [2, 4].
- **npcols:** Integer range of [2, 4].

As in Section 4.4, the experiments were conducted on the 2x AMD EPYC 9654 processors that consist of a total of 192 (2x96) physical cores and 2 hyperthreads per core. We mainly considered the Si5H12 sparse matrix obtained from the Suite Sparse matrix collection [13]. This application was executed with OpenMPI 4.1.5.

Experiments and results. For a performance variability identifier kernel, we intercept DGEMM calls made during the sample execution. Since SuperLU_DIST invokes thousands of DGEMM calls, we apply sampling to reduce instrumentation overhead. DGEMM is a stable compute-bound kernel whose runtime depends only on the matrix dimensions, making it a good indicator of variability under runtime compute noise. For workload identification, we capture the performance counter total instruction count using PAPI. We then compute the variability score using the formula detailed in Section 3. Compute noise was injected every 3rd run.

Figure 6 shows the tuning performance on SuperLU_DIST using the Si5H12 matrix. With 50 evaluations, VaBO achieves a final objective runtime value of 6.59s, approximately 4% improvement over

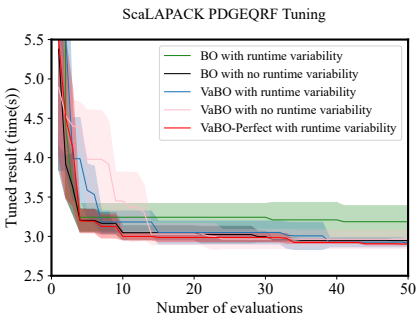


Figure 5: Tuning performance for ScaLAPACK's PDGEQRF, with input size of $m = n = 10,000$. Mean of 3 runs; shaded area denotes standard deviation.

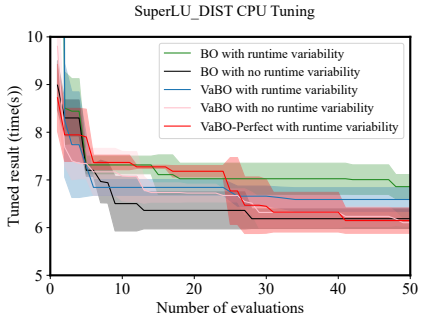


Figure 6: Tuning results for SuperLU_DIST matrix Si5H12 [13]. Mean of 3 runs; shaded area denotes standard deviation.

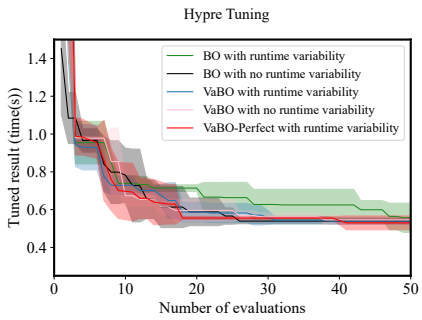


Figure 7: Tuning performance for Hypre IJ Solver time on a fixed-sized matrix, $nx = ny = nz = 125$. Mean of 3 runs; shaded area denotes standard deviation.

the 6.86s by standard BO under the same noisy conditions. However, as seen in the plot, more importantly, VaBO converges clearly faster than regular BO tuning under compute runtime variability.

4.6 Hypre

Experimental setup. The tuning scenario is tuning for Hypre, a parallel multigrid solver for sparse linear systems. Particularly, we consider tuning for Hypre AMG Solver on a fixed-size matrix generated by the IJ application. This tuning problem consists of many more parameters compared to the experiments in the previous sections. Our tuning parameters include the following.

- **Px:** Integer range of [1, 12]
- **Py:** Integer range of [1, 12]; Px and Py are used for processor topology.
- **Nproc:** Integer range of [1, 12]
- **strong_threshold** Real range of [0, 1]; strength threshold.
- **coarsen_type** Categorical options of [0, 1,2,3,4,6,8,10]; coarsening algorithm.
- **relax_type** Categorical options of [-1, 0,6,8,16,18]; smoother option.
- **interp_type** Categorical options of [0, 2,4,6,8,12]; interpolation operator.

As in Section 4.4, the experiments were conducted on the 2x AMD EPYC 9654 processors that consist of a total of 192 (2x96) physical cores and 2 hyperthreads per core.

Experiments and results. The solve phase of the IJ routine repeatedly calls the `hypre_ParCSRMatrixMatvec` kernel with fixed matrix and vector sizes. The `hypre_ParCSRMatrixMatvec` kernel performs sparse matrix-vector multiplication (SpMV). Since SpMV is memory-bound, and has fixed matrix sparsity during the solve phase of IJ, its performance is highly sensitive to memory bandwidth contention. To measure system variability under memory noise, we intercept the MPI_Send calls invoked during parallel SpMV operation. We identify the workload by categorizing these communication calls into bins based on their message size to ensure that variability is calculated across workload sizes. We measure the latency of these intercepted calls and compute a variability score as per Section 3.3.

For noise scenarios, we introduce memory interference to contend with the memory bound SpMV kernel. The noise processes are pinned to specific cores (hyperthreads of the application processes). The noise processes perform STREAM Triad like memory operations with random short sleep and burst intervals designed to saturate memory bandwidth and cause significant contention. The noise was injected deterministically for every fifth run.

Figure 7 shows the tuning results. With 50 evaluations, we see VaBO achieves a final objective runtime value of 0.53s, approximately 4% improvement over the 0.56s by standard BO under the same noisy conditions. Although the improvements are more marginal in this scenario compared to previous experiments (e.g., the ScaLAPACK example), VaBO still provides robust tuning quality, consistently outperforming standard Bayesian Optimization (BO) with runtime variability.

4.7 Accuracy of variability detection tool

To evaluate the accuracy of the variability detection tool, we performed tests across the applications SuperLU_DIST (Section 4.5), Hypre (Section 4.6), ScaLAPACK PDGEQRF (Section 4.4), and the neural network inference (Section 4.3).

For overhead analysis, variability was added deterministically to the runs. We assessed the tool's detection performance using standard classification metrics, including True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). These metrics were also used to compute an overall accuracy value in each tested application.

The results of these evaluations are shown in Table 1. We observe Hypre and Neural Network Inference show strong detection performance (90.67% and 83.9% accuracy). In contrast, ScaLAPACK has lower accuracy, with more false positives and some false negatives. This could be attributed to higher inherent variability due to complex MPI communication. Overall, from the tuning results, we find that achieving very high accuracy is not strictly necessary for VaBO to be effective.

4.7.1 Noise Sensitivity Results. To assess noise sensitivity, we record the variability scores of ScaLAPACK's PDGEQRF application under different noise levels. Artificial noise is injected via background

Table 1: Accuracy of the runtime variability detection tool across applications. Each instance is one application run. Variability identifier kernels captured during the run are used to compute an aggregate runtime variance, which is classified as exhibiting variability (positive) if it exceeds the threshold. TP/FP/TN/FN are computed against the known ground truth.

Application	TP	FP	TN	FN	Accuracy
Neural Network	52	19	131	16	83.9%
SuperLU_DIST (CPU)	90	12	271	49	85.5%
ScaLAPACK	17	45	78	5	65.5%
Hypre (IJ)	23	4	113	10	90.67%

stressor processes at 4 distinct intensities (0, 4, 8, and 12 processes), corresponding to 0%, 33%, 66%, and 100% of core occupancy relative to the application's configuration (which uses 12 cores for this experiment).

Figure 8 illustrates the relationship between noise sensitivity and measured variability score. We observe a clear increase in the variability score as the number of noise processes increase. At the baseline (0 noise processes), the application exhibit a low variability score of approximately 0.2. As the interference increases, the score grows steadily, saturating near 1.0 when the number of noise processes equals the number of application process workers.

4.8 Overhead Analysis

To assess the performance impact of the variability detection interceptor, we measured the execution runtime of the SuperLU_DIST CPU application with and without the VaBO library enabled.

As shown in Table 2, the overhead introduced by the interceptor is small, and becomes negligible as the workload size increases. We see that for SuperLU_DIST, the overhead dropped from 10.4% on millisecond-scale tasks (Si2) to 0.16% for the Si5H12 matrix. Similarly, Hypre showed a significant reduction from 3.45% to 0.35% as the input matrix size grows. We see that ScaLAPACK also demonstrates low overhead remaining below 1.3% across all matrix sizes. These results confirm that our interception-based variability detection is sufficiently lightweight to be integrated into Bayesian Optimization tuning without compromising application performance.

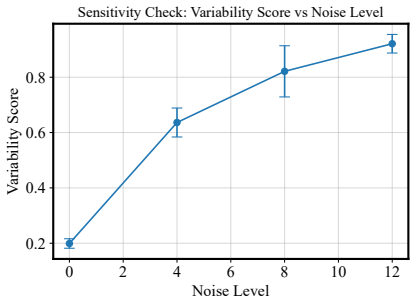


Figure 8: Noise Sensitivity Results for ScaLAPACK PDGEQRF.

Table 2: Interceptor overhead across problem sizes.

Workload	Baseline (No Int)	VaBO (With Int)	Overhead (%)
SuperLU_DIST			
Si2	0.0048 s	0.0054 s	~10.416%
SiH4	0.5482 s	0.5677 s	~3.434%
Si10H16	8.256 s	8.269 s	~0.359%
Si5H12	18.9294 s	18.9977 s	~0.157%
ScaLAPACK			
5000	0.54 s	0.55 s	~0.851%
10000	3.13 s	3.14 s	~0.3194%
15000	9.13 s	9.25 s	~1.297%
20000	20.93 s	21.11 s	~0.860%
Hypre			
100	0.4384 s	0.4536 s	~3.45%
125	0.8819 s	0.8879 s	~0.6768%
150	1.645 s	1.6540 s	~0.5449%
175	2.2035 s	2.2163 s	~0.5772%
200	3.3047 s	3.3162 s	~0.3459%

4.9 GPU Case Study

Compared to CPUs, GPU runtime variability can happen due to the heat dissipation, rather than OS noise or shared resource contention. However, we envision that as the GPU systems scale to execute more and more heavy workloads, the variability would become a critical issue. Hence, to show the applicability and the potential of variability-aware tuning we conduct more experiments on GPUs.

For this set of experiments, we utilized NVIDIA Tesla V100 GPU equipped with 32 GB of HBM2 memory, and AMD Instinct MI210 equipped with 64 GB of HBM2e memory. For GPU-based evaluation, we focus on *SuperLU_DIST* configured to use the `pddrive3d` algorithm. The 3D algorithm is designed to efficiently use GPU resources, and is hence well-suited for the GPU-accelerated execution of the SuperLU_DIST solver.

Since it is difficult to inject controlled GPU interference from OS scheduling levels, we model runtime variability by imposing a slowdown on every other execution relative to baseline runs. We artificially simulate runtime noise by scaling the reported execution time of the application by a factor whenever noise occurs. Each re-evaluation of a parameter configuration increments the evaluation run index. As a representative evaluation scenario, we induce a controlled 2× runtime increase occurring in 50% of NVIDIA GPU runs and 25% of AMD GPU runs.

Given our synthetic noise model, we limit our comparison to the tuners BO_with_no_variability, BO_with_runtime_variability and VaBO_Perfect_with_runtime_variability, since our detection tool relies on actual performance counters which are not applicable to artificial slowdowns.

From Table 3, we observe the performance of VaBO against the BO+var across different matrix sizes for the 2× slowdown configuration. We observe that the VA_tuner consistently performs better than or equal to BO under runtime variability for both GPUs.

To provide a sensitivity analysis across different noise patterns, we vary the noise interval and intensity of the injected variability on the AMD GPU. Here, noise interval k denotes that noise is

Table 3: Best runtime (s) of SuperLU_DIST on GPUs

NVIDIA GPU

Matrix	Tuner	10 evals	20 evals	30 evals	40 evals	50 evals
SiNa	BO	0.375	0.333	0.327	0.323	0.320
SiNa	BO+var	0.415	0.390	0.375	0.365	0.355
SiNa	VaBO_Perf	0.340	0.335	0.332	0.330	0.329
Si5H12	BO	3.00	2.80	2.74	2.72	2.67
Si5H12	BO+var	3.35	3.18	3.10	3.00	2.93
Si5H12	VaBO_Perf	2.84	2.82	2.80	2.78	2.76
Si10H16	BO	2.12	2.08	2.06	2.06	2.05
Si10H16	BO+var	2.17	2.12	2.10	2.08	2.07
Si10H16	VaBO_Perf	2.11	2.08	2.07	2.06	2.05

AMD GPU

Matrix	Tuner	10 evals	20 evals	30 evals	40 evals	50 evals
SiNa	BO	0.532	0.395	0.395	0.395	0.395
SiNa	BO+var	0.580	0.442	0.395	0.395	0.395
SiNa	VaBO_Perf	0.532	0.454	0.395	0.395	0.395
Si5H12	BO	2.966	2.848	2.848	2.848	2.848
Si5H12	BO+var	3.055	3.055	2.848	2.848	2.848
Si5H12	VaBO_Perf	2.966	2.966	2.848	2.848	2.848
Si10H16	BO	2.470	2.362	2.362	2.242	2.223
Si10H16	BO+var	2.468	2.371	2.371	2.371	2.360
Si10H16	VaBO_Perf	2.470	2.362	2.362	2.362	2.340

BO+var = BO under variability; VaBO_Perf = VaBO-Perfect under variability

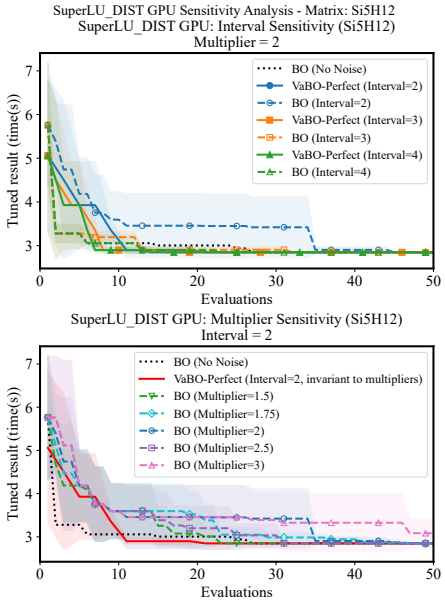


Figure 9: Comparison of VaBO_Perfect and BO with runtime variability under different noise patterns (noise interval and multiplier (noise intensity)) on the AMD GPU for the SuperLU_DIST tuning problem. Mean of 3 runs; shaded area denotes standard deviation.

injected every k-th evaluation, and intensity represents the multiplier applied to the execution time. Prior studies have observed substantial performance variability on GPUs, including up to 1.5× slowdown on V100 GPUs [42], up to 3.5×variability on V100 GPUs [29], and up to 3.2× slowdown on production supercomputers [49]. These observations support the feasibility of our synthetic noise scenarios.

In Figure 9, we evaluate sensitivity to noise intervals and slowdown multipliers. First, we fix the noise multiplier to 2× and vary the injection interval across 2, 3, and 4 evaluations. As seen in the figure, the most frequent noise pattern (Interval = 2) has the largest impact on standard BO, leading to slower convergence and a larger performance gap between BO and VaBO_Perfect. As the interval increases to 3 and 4, the noise occurs less frequently and the performance gap decreases. However we observe that VaBO_Perfect continues to converge faster than regular BO across all three intervals. Next, we fix the noise injection interval at 2 evaluations and vary the slowdown multiplier from 1.5× to 3×. We observe the trend that stronger noise increasingly slows the convergence of regular BO. The performance improvement of VaBO_Perfect becomes more prominent as the slowdown multiplier increases. These results indicate that VaBO is most beneficial when runtime variability is both frequent and severe.

4.10 Experiments with More Noise Scenarios

The earlier experimental results (from Section 4.3 to 4.9) considered the synthetic noises introduced in Section 4.1. In contrary, this section evaluates VaBO under two additional kinds of noise: real interference from co-running applications (Section 4.10.1), and emulated interference from OS background processes (Section 4.10.2). Section 4.10.3 discusses the limitations and implications of the experiments.

4.10.1 Co-running Application as Noise. Here, we run experiments of VaBO, in which HPC/parallel applications (ScaLAPACK, Hypre, SuperLU_DIST) act as co-running workloads while VaBO tunes a given target application, under two allocation policies: (i) time-shared, and (ii) space-shared. In time-shared allocation, both applications run on the same cores and the OS scheduler arbitrates between them, which adds CPU contention and scheduling delays on top of the shared-resource interference. In space-shared allocation, the application under tuning and the co-running application are pinned to disjoint sets of cores, so that interference arises through shared resources (memory bandwidth, last-level cache) rather than through CPU time.

Our experiments focus on tuning results for SuperLU_DIST with ScaLAPACK running as the co-running workload, since this setting produced enough slowdown to show the improvement of VaBO over regular BO. For the other tuning application-corunning application pairings, the induced interference was not enough to cause slowdowns in the sample evaluations. VaBO is designed to perform on-par with BO under no-noise scenarios and to show improvement in unreliable conditions. We confirmed that behavior for the other applications in preliminary results, and hence focus our tuning analysis on the SuperLU_DIST - ScaLAPACK pairing for this noise model.

Figure 10 (top) shows the best configuration result found over tuning iterations for SuperLU_DIST (Si5H12 matrix) when co-running with ScaLAPACK in the background in the time-shared setting, as a case study among the application combinations evaluated. A total

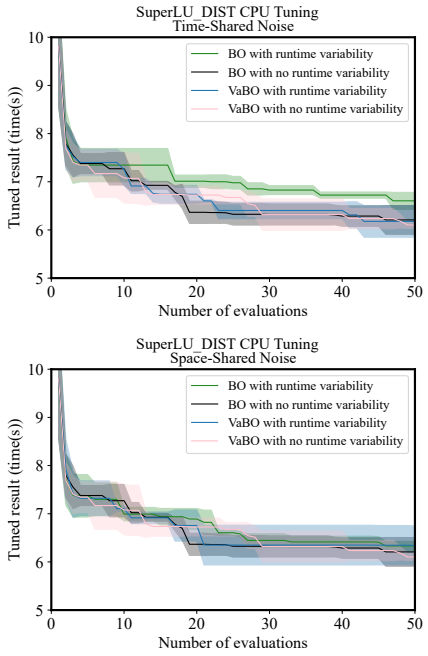
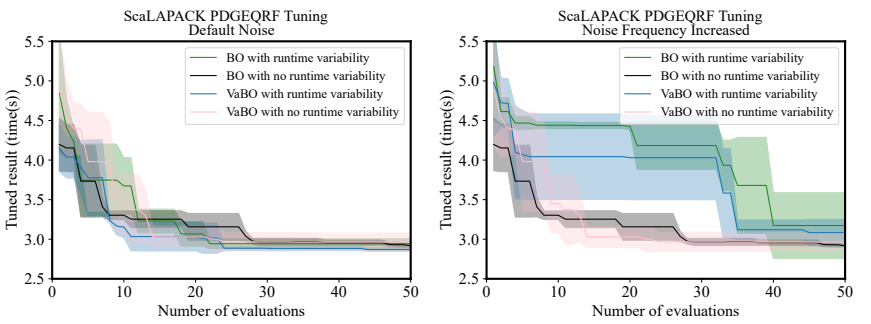


Figure 10: Tuning quality under co-running applications. Mean of 3 runs; shaded area denotes standard deviation.



Daemon and its original activities	Simplified proxy	Interval (default; from [31])	Interval (shortened)
`cerebrod`: Heartbeat multicast; UDP payload serialization	CPU spin	10–20 s	5–10 s
`irqbalance`: Reads interrupt table and computes CPU IRQ reassignments	File read + parsing ops + CPU spin	10 s	5 s
`khugepaged`: Walks memory mappings (4,096 pages) to collapse transparent huge pages	CPU spin	10 s	5 s
`ldmsd`: Metric sampler; parses and serializes `meminfo`, `vmstat`, `stat`	File read + serialization ops + CPU spin	1.0 s	0.5 s

Figure 11: Tuning quality with emulated OS noise. Results averaged over 3 runs. The table here summarizes four primary OS noise sources identified in [31] that we emulate using simplified proxy functions, along with their wake intervals in both the default and frequency-increased configurations. Wake interval of 10-20s refers to a random selected value between 10 and 20s.

of 30.7% of evaluations (46 out of 150 samples across 3 iterations) experienced noise-induced runtime slowdown. The variability detection module triggered re-evaluations for these unstable configurations, with approximately 60.9% of second-run executions being faster. We see that VaBO performs better than BO under this noise setting. Under the space-shared memory noise setting as seen in Figure 10 (bottom), core pinning isolated execution such that memory contention alone was insufficient to cause runtime variance in SuperLU_DIST. We observe that the variability detection threshold was never exceeded, reruns were not triggered, and VaBO performed similar to standard BO.

4.10.2 Emulated OS background processes as Noises. León et al. [31] studied the noise impact on large-scale computers, particularly how OS services can become critical runtime variability in large-scale computers. From measurement on supercomputers including El Capitan, they identify the OS processes that contribute most, including `cerebrod`, `irqbalance`, `khugepaged`, and `ldmsd`. Here, we used the characterization from [31] to construct noise scenarios that reproduce the periodic wake intervals and disruption patterns of the top OS-level noise sources. Two aspects of our setting required adapting the scenarios. First, the OS noise studied becomes critical at large-scale (supercomputer class), whereas our evaluation of VaBO runs on single-node systems, where the same OS background/interrupt processes are ordinary background activity.

We evaluated two variants: (a) the noise sources with the intervals and durations reported in the paper or process defaults, and (b) a variant with a higher intensity in which the intervals are compressed to increase the event density within the benchmark window, approximating the cumulative noise that a longer or larger-scale execution would experience. Second, the processes the paper identifies are OS services, and controlling them requires superuser access, which we do not have in our experimental environment. We therefore emulate each service with a rather simplified, user-space proxy process that reproduces its stalling frequency pattern.

We find that this noise scenario creates the unreliable conditions setting that VaBO is designed to perform best, most prominently for the ScaLAPACK tuning problem. For the other applications (SuperLU_DIST and Hypre) we find that the noise does not create sufficient slowdown and unreliability, and hence VaBO performs similarly to BO. In our preliminary experiments for the other applications, we confirmed this behavior. For this experiment, we focus on and report tuning for ScaLAPACK under this noise condition.

Figure 11 (left) shows the best configuration result over tuning iterations for ScaLAPACK under a default OS noise setting. 43.3% of the sample evaluations (65 out of 150 evaluations across 3 iterations) were found to experience variance. Figure 11 (right) shows best configuration result over tuning iterations under the higher intensity OS noise setting. 46.0% (69 out of 150 evaluations across 3 iterations) were found to be unreliable by the variability detection module and triggered reruns. 50.7% (35/69) of the second run executions were found to be faster. VaBO is seen to perform better than BO under the noise settings.

4.10.3 Limitations and Discussions. The noise from a co-running application in a space partitioned allocation and/or OS-level processes within a single node scale is rather ordinary noise than being critical to impact huge runtime variability of a sample. Yet,

time-shared allocation and/or frequency-increased noise from OS process impacted the runtime and VaBO became more effective.

We emphasize that VaBO is designed to perform reliable BO-based autotuning for parallel applications under unreliable conditions without assuming knowledge of the noise pattern, focusing on HPC applications. This paper does not claim to model and reproduce a specific real noise pattern, which would vary across systems and is out of this paper's scope. Our experiments are also limited to a single node scale. Although our HPC cluster also has intrinsic noise, that intrinsic noise is not critical at this scale. Nevertheless, the community has found that the noise increases and becomes significant as the system scale grows [29, 31]. Therefore, despite the limitations that intrinsic noise is relatively small in our testbed, VaBO's contribution on reliable sampling in BO-based performance tuning will provide a foundation for reliable tuning for larger scales.

5 Related Work

Empirical autotuning and Bayesian optimization (BO). Pioneering empirical autotuning frameworks, such as ATLAS [10] and PHiPAC [6], laid the foundation for optimizing dense linear algebra kernels and matrix multiply workloads such as GEMM. Oski [48] extended autotuning to sparse matrix computations. Later several tools such as OpenTuner [2] were presented as a general autotuner that employs various search methodologies. With the advent of the exascale era and the increasing cost of running HPC applications, the field has shifted towards data-driven, sample-efficient methods, where Bayesian optimization (BO) is widely adopted. Prominent BO-based autotuners for HPC include GPTune [33], ytopt [52], BLISS [41], and HiPerBOt [39]. GPTune [33] distinguishes itself by employing multitask learning and distributed-parallel surrogate modeling, and ytopt was applied to tune various code sections, and BLISS employs a pool of learning models. The BO methodology has also been applied to a GPU kernel tuning framework called KernelTuner [47, 50].

Transfer learning and advanced surrogate modeling in BO. A complementary line of work in BO tuning has focused on further reducing tuning cost via reuse and transfer. [9] aggregates performance data across users and environments to accelerate BO-driven search under tight budgets. HiPerBOt[39] also uses transfer learning to learn from smaller problems to larger problems. Randall et al. [40] propose a Gaussian copula model that generates high-performing configurations from prior data for transfer learning. On the other hand, [35] addresses non-smooth objective surfaces using clustered Gaussian processes, and [36] proposes hybrid parameter search with dynamic model selection for mixed-variable BO. Beyond sequential BO, speculation-based strategies can parallelize evaluations to better utilize HPC resources and increase throughput of BO search [14].

Performance variability and robust tuning. While autotuning methodologies have matured, performance variability remains a critical challenge in HPC environments. Several existing works such as vSensor [45], Vapro [53], and Su et al. [43, 44] have explored techniques to detect and pinpoint performance variability and/or inefficiencies. On the other hand, Trümper et al. [46] introduce performance embeddings that map static and dynamic program features into a continuous space, enabling nearest-neighbor transfer of optimization records to reduce search space and accelerate tuning. On the I/O side, Costa et al. [11] analyze I/O performance variability on production HPC systems and show that even repetitive I/O patterns can exhibit substantial runtime fluctuations due to workload characteristics. And Behzad et al. [5] develop a genetic search-based autotuner for parallel HDF5 I/O that intercepts I/O calls and adaptively selects filesystem parameters, achieving large performance gains despite highly variable I/O behavior.

Differentiation from previous work. The aforementioned works are largely orthogonal to VaBO and can be combined with VaBO for enhanced performance, as they do not perform online detection of unstable evaluations nor prevent corrupted samples from influencing surrogate models. In contrast, VaBO directly incorporates variability awareness into Bayesian optimization by identifying and handling unstable measurements during tuning, enabling robust, reliable autotuning in noisy execution environments.

6 Conclusion

This work shows that runtime performance variability fundamentally limits the effectiveness of Bayesian optimization-based autotuning on modern performance critical computing systems. We introduce VaBO, a variability-aware tuning framework that detects unstable evaluations online and prevents unreliable samples from corrupting surrogate models.

Across HPC benchmarks such as SuperLU_DIST, ScaLAPACK PDGEQRF, Hypre IJ, and a neural network inference, VaBO converges faster than standard BO under compute and memory noise and achieves better best found runtime for most scenarios. From our experimental results, we observe that VaBO finds 4-8% better tuned result within a tuning budget of 50 evaluations compared to baseline BO under runtime variability. At scale, these improvements can represent significant savings. The variability detector identifies unstable samples with 65–90% accuracy in complex variability environments across applications and exhibits monotonic sensitivity to increasing noise levels.

These results demonstrate that ignoring variability is no longer tenable for empirical autotuning. Instead, tuners must explicitly reason about noise as a critical property of the optimization process. VaBO provides a practical, lightweight path to noise-robust autotuning.

Acknowledgments

We thank our shepherd, Sreepathi Pai, and the anonymous reviewers for their valuable feedback. We also thank the other members of the GPTune team for helpful discussions over the years that inspired the problem identification of this work. This work was supported in part by Santa Clara University and Rice University. Most experiments were conducted on the WAVE HPC cluster at Santa Clara University. The AMD GPU experiments used computing resources made available through the AMD University Program (AUP) AI & HPC Cluster. Saeyeon Kim was in part supported by a Clare Boothe Luce (CBL) Grant from the Henry Luce Foundation as a student scholar. Generative AI tools were used for general coding and writing assistance; all technical content and conclusions are the authors' original work.

References

[1] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK users' guide.* SIAM.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation.* 303–316.

[3] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (2018), 2068–2083. doi:10.1109/JPROC.2018.2841200

[4] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2019. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Trans. Parallel Comput.* 5, 4, Article 15 (March 2019), 27 pages. doi:10.1145/3309205

[5] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. 2013. Taming parallel I/O complexity with auto-tuning. In *Proceedings of the international conference on high performance computing, networking, storage and analysis.* 1–12.

[6] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (Munich, Germany). Association for Computing Machinery, New York, NY, USA, 253–260. doi:10.1145/2591635.2667174

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898719642

[8] Younghyun Cho, James W. Demmel, Grace Dinh, Xiaoye S. Li, Yang Liu, Hengrui Luo, Osni Marques, and Wissam M. Sid-Lakhdar. 2022. GPTune User Guide. https://gptune.lbl.gov/documentation/gptune-user-guide. https://gptune.lbl.gov/documentation/gptune-user-guide

[9] Younghyun Cho, James W. Demmel, Jacob King, Xiaoye S. Li, Yang Liu, and Hengrui Luo. 2023. Harnessing the Crowd for Autotuning High-Performance Computing Applications. In *The IEEE 37th International Parallel and Distributed Processing Symposium (IPDPS).*

[10] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35. doi:10.1016/S0167-8191(00)00087-9 New Trends in High Performance Computing.

[11] Emily Costa, Tirthak Patel, Benjamin Schwaller, Jim M. Brandt, and Devesh Tiwari. 2021. Systematically inferring I/O performance variability by examining repetitive job behavior. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21).* Association for Computing Machinery, New York, NY, USA, Article 33, 15 pages. doi:10.1145/3458817.3476186

[12] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. doi:10.1109/99.660313

[13] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. doi:10.1145/2049662.2049663

[14] Adrian P Dieguez, Seth Ockerman, Tristan Aikman, Younghyun Cho, Yang Liu, and Khaled Z Ibrahim. 2025. Parallelizing autotuning for HPC applications: Unveiling the potential of the speculation strategy in Bayesian optimization. *The International Journal of High Performance Computing Applications* 39, 5 (2025), 631–654. doi:10.1177/10943420251362001

[15] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim Yarkhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Šístek, David Stevens, Mawussi Zounon, and Samuel D. Relton. 2019. PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP. *ACM Trans. Math. Softw.* 45, 2, Article 16 (may 2019), 35 pages. doi:10.1145/3264491

[16] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16, 1 (mar 1990), 1–17. doi:10.1145/77626.79170

[17] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, Eduardo Cesar, Anna Sikora, and Ali Jannesari. 2023. Performance Optimization using Multimodal Modeling and Heterogeneous GNN. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23).* Association for Computing Machinery, New York, NY, USA, 45–57. doi:10.1145/3588195.3592984

[18] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, Eduardo Cesar, Anna Sikora, and Ali Jannesari. 2023. Performance Optimization using Multimodal Modeling and Heterogeneous GNN. In *HPDC 2023-32nd International Symposium on High-Performance Parallel and Distributed Computing.*

[19] Akash Dutta, Jee Choi, and Ali Jannesari. 2023. Power Constrained Autotuning using Graph Neural Networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 535–545. doi:10.1109/IPDPS54959.2023.00060

[20] Robert D. Falgout and Ulrike Meier Yang. 2002. hypre: A Library of High Performance Preconditioners. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 632–641.

[21] Rob Farber. 2011. *CUDA application design and development.* Elsevier.

[22] Peter I Frazier. 2018. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018).

[23] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. 2008. MILEPOST GCC: machine learning based research compiler. In *GCC Summit.* Ottawa, Canada. https://hal.inria.fr/inria-00294704

[24] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–104.

[25] Robert B Gramacy. 2020. *Surrogates: Gaussian process modeling, design, and optimization for the applied sciences.* OCLC: 1120696204.

[26] William Gropp and Ewing Lusk. 1996. User's Guide for mpich, a Portable Implementation of MPI.

[27] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–11. doi:10.1109/SC.2010.12

[28] Heike Jagode, Anthony Danalis, Hartwig Anzt, and Jack Dongarra. 2019. PAPI software-defined events for in-depth performance analysis. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1113–1127.

[29] Rutwik Jain, Brandon Tran, Keting Chen, Matthew D. Sinclair, and Shivaram Venkataraman. 2024. PAL: A Variability-Aware Policy for Scheduling ML Workloads in GPU Clusters. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–18. doi:10.1109/SC41406.2024.00032

[30] Hong Jia-Wei and H. T. Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (Milwaukee, Wisconsin, USA) *(STOC '81).* Association for Computing Machinery, New York, NY, USA, 326–333. doi:10.1145/800076.802486

[31] Edgar A. León, Joseph Glenski, Mark Stock, Kim McMahon, William Loewe, Clark Snyder, Larry Kaplan, Srinath Vadlamani, Timothy I. Mattox, Trent D'Hooge, Brian Behlendorf, Nathan Hanford, Ramesh Pankajakshan, and Matthew L. Leininger. 2025. Breaking the System Noise Barrier at Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25).* Association for Computing Machinery, New York, NY, USA, 411–436. doi:10.1145/3712285.3759793

[32] Xiaoye S. Li and James W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.* 29, 2 (June 2003), 110–140. doi:10.1145/779359.779361

[33] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. 2021. GPTune: Multitask Learning for Autotuning Exascale Applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21).* Association for Computing Machinery, New York, NY, USA, 234–246. https://doi.org/10.1145/3437801.3441621

[34] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 2010. Managing Variability in the IO Performance of Petascale Storage Systems. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–12. doi:10.1109/SC.2010.32

[35] Hengrui Luo, Younghyun Cho, James W Demmel, Igor Kozachenko, Xiaoye S Li, and Yang Liu. 2024. Non-smooth Bayesian optimization in tuning scientific applications. *The International Journal of High Performance Computing Applications* 38, 6 (2024), 633–657. doi:10.1177/10943420241278981

[36] Hengrui Luo, Younghyun Cho, James W. Demmel, Xiaoye S. Li, and Yang Liu. 2024. Hybrid Parameter Search and Dynamic Model Selection for Mixed-Variable Bayesian Optimization. *Journal of Computational and Graphical Statistics* 33, 3 (2024), 855–868. doi:10.1080/10618600.2024.2308216

[37] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[38] John D. McCalpin. 2018. HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* 225–237. doi:10.1109/SC.2018.00021

[39] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. 2020. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 831–840. doi:10.1109/IPDPS47924.2020.00090

[40] Thomas Randall, Jaehoon Koo, Brice Videau, Michael Kruse, Xingfu Wu, Paul Hovland, Mary Hall, Rong Ge, and Prasanna Balaprakash. 2023. Transfer-learning-based Autotuning using Gaussian Copula. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 37–49. doi:10.1145/3577193.3593712

[41] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2021. *Bliss: Auto-Tuning Complex Applications Using a Pool of Diverse Lightweight Learning Models*. Association for Computing Machinery, New York, NY, USA, 1280–1295. https://doi.org/10.1145/3453483.3454109

[42] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D. Sinclair, and Shivaram Venkataraman. 2022. 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*. 01–15. doi:10.1109/SC41404.2022.00070

[43] Pengfei Su, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2019. Pinpointing performance inefficiencies via lightweight variance profiling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 19 pages. doi:10.1145/3295500.3356167

[44] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant Loads: A Software Inefficiency Indicator. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 982–993. doi:10.1109/ICSE.2019.00103

[45] Xiongchao Tang, Jidong Zhai, Xuehai Qian, Bingsheng He, Wei Xue, and Wenguang Chen. 2018. vSensor: leveraging fixed-workload snippets of programs for performance variance detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 124–136. doi:10.1145/3178487.3178497

[46] Lukas Trümper, Tal Ben-Nun, Philipp Schaad, Alexandru Calotoiu, and Torsten Hoefler. 2023. Performance embeddings: A similarity-based transfer tuning approach to performance optimization. In *Proceedings of the 37th International Conference on Supercomputing*. 50–62.

[47] Ben van Werkhoven. 2019. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 90 (2019), 347–358. doi:10.1016/j.future.2018.08.004

[48] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16, 1 (jan 2005), 521. doi:10.1088/1742-6596/16/1/071

[49] Cunyang Wei, Keshav Pradeep, and Abhinav Bhatele. 2025. Unmasking Performance Variability in GPU Codes on Production Supercomputers. Research poster, SC25: The International Conference for High Performance Computing, Networking, Storage, and Analysis, St. Louis, MO, USA. https://sc25.supercomputing.org/proceedings/posters/poster_files/post256s2-file3.pdf

[50] Floris-Jan Willemsen, Rob van Nieuwpoort, and Ben van Werkhoven. 2021. Bayesian Optimization for auto-tuning GPU kernels. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 106–117. doi:10.1109/PMBS54543.2021.00017

[51] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[52] Xingfu Wu, Prasanna Balaprakash, Michael Kruse, Jaehoon Koo, Brice Videau, Paul Hovland, Valerie Taylor, Brad Geltz, Siddhartha Jana, and Mary Hall. 2025. ytopt: Autotuning Scientific Applications for Energy Efficiency at Large Scales. *Concurrency and Computation: Practice and Experience* 37, 1 (2025), e8322. doi:10.1002/cpe.8322

[53] Liyan Zheng, Jidong Zhai, Xiongchao Tang, Haojie Wang, Teng Yu, Yuyang Jin, Shuaiwen Leon Song, and Wenguang Chen. 2022. Vapro: performance variance detection and diagnosis for production-run parallel applications. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 150–162. doi:10.1145/3503221.3508411