Bootstrapping Parameter Space Exploration for Fast Tuning

Jayaraman J. Thiagarajan*
Lawrence Livermore National Laboratory
jayaramanthi1@llnl.gov

Nikhil Jain†
Lawrence Livermore National Laboratory
nikhil@llnl.gov

Rushil Anirudh
Lawrence Livermore National Laboratory
anirudh1@llnl.gov

Alfredo Gimenez
Lawrence Livermore National Laboratory
gimenez1@llnl.gov

Rahul Sridhar
University of California, Irvine
rsridha2@uci.edu

Aniruddha Marathe
Lawrence Livermore National Laboratory
marathe1@llnl.gov

Tao Wang
North Carolina State University
twang15@ncsu.edu

Murali Emani
Lawrence Livermore National Laboratory
emani1@llnl.gov

Abhinav Bhatele
Lawrence Livermore National Laboratory
bhatele@llnl.gov

Todd Gamblin
Lawrence Livermore National Laboratory
gamblin2@llnl.gov

ABSTRACT

The task of tuning parameters for optimizing performance or other metrics of interest such as energy, variability, etc. can be resource and time consuming. Presence of a large parameter space makes a comprehensive exploration infeasible. In this paper, we propose a novel bootstrap scheme, called GEIST, for parameter space exploration to find performance-optimizing configurations quickly. Our scheme represents the parameter space as a graph whose connectivity guides information propagation from known configurations. Guided by the predictions of a semi-supervised learning method over the parameter graph, GEIST is able to adaptively sample and find desirable configurations using limited results from experiments. We show the effectiveness of GEIST for selecting application input options, compiler flags, and runtime/system settings for several parallel codes including LULESH, Kripke, Hypre, and OpenAtom.

CCS CONCEPTS

• **General and reference → Performance**; • **Theory of computation → Semi-supervised learning**; • **Computing methodologies → Search with partial observations**;

KEYWORDS

autotuning, sampling, performance, semi-supervised learning

ACM Reference Format:
Jayaraman J. Thiagarajan, Nikhil Jain, Rushil Anirudh, Alfredo Gimenez, Rahul Sridhar, Aniruddha Marathe, Tao Wang, Murali Emani, Abhinav Bhatele, and Todd Gamblin. 2018. Bootstrapping Parameter Space Exploration for Fast Tuning. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205321

1 INTRODUCTION

As the complexity of High-Performance Computing (HPC) and big-data systems, software stacks, and applications continue to rise, achieving high performance has become difficult. Most components of these ecosystems are increasingly becoming more configurable, and to maximize performance, correctly configuring these components has become essential. To illustrate this concern, Figure 1 shows the distribution of runtime for Kripke [21], a transport code, with different configurations. Here, performance varies by 1000x depending on the choice of application parameter values for a constant input problem.

The number of tunable parameters that a user can configure has increased linearly, and as a result, the overall parameter space has grown exponentially. In addition, optimizing for performance metrics other than execution time, such as energy consumption, has become increasingly essential[1]. Exhaustively evaluating parameter combinations for these different dependent variables is intractable, and hence automatic exploration of parameter space, called *autotuning*, is desirable.

*J.J. Thiagarajan and N. Jain contributed equally to this work
†The corresponding author

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205321

[1]Throughout this paper, we use "performance" as a generic term to refer to the metric being optimized, such as execution time, energy, and variability.

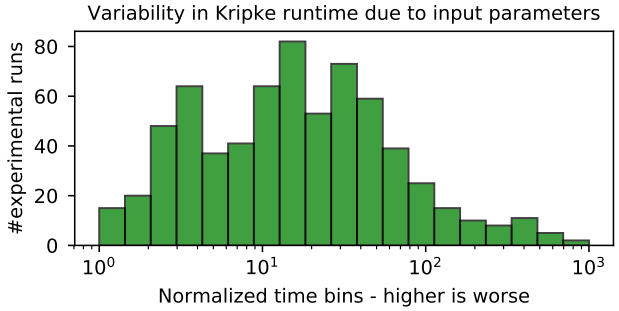


Figure 1: Sub-optimal choice of configuration can result in up to 1000× slowdown for a constant input problem.

Autotuning requires quantifying the effects that different parameters will have on performance. However, making this determination *a priori* is usually infeasible, as it would require constructing complex models for a variety available parameters and system environments. Therefore, autotuning frameworks typically employ empirical approaches by collecting performance samples and adjusting a model to fit them. However, collecting a large number of performance samples can be prohibitively expensive as individual runs may take minutes to hours to complete. Autotuning therefore requires methods to automatically reduce the search space of possible configurations to avoid expensive training while retaining enough information to determine performance-optimizing configurations.

Traditional methods for autotuning are typically built upon heuristics that derive from experience [9, 14]. Many of these methods often need to be reworked as new parameters become available. Further, several existing approaches utilize simple prediction techniques such as linear regression, and hence require a reasonably large number of samples for better decision making. Recent work has shown promise in the use of sophisticated statistical learning techniques to build accurate and generalizable models, thus reducing the overheads of autotuning [23, 26]. In particular, *adaptive sampling*, a technique in which sample collection is performed incrementally, has produced encouraging results [10].

In this paper, we develop a new approach to minimize the number of samples being collected in order to identify high-performing configurations, while minimizing the time spent in exploring sub-optimal configurations. Our approach, named *Good Enough Iterative Sampling for Tuning* (GEIST), uses semi-supervised learning to effectively guide the search of high-performing configurations, while being robust to the choice of the initial sample set.

Specifically, this paper makes the following contributions:

- We introduce GEIST, a novel semi-supervised learning-based adaptive sampling scheme for parameter space exploration.
- We show that GEIST finds performance optimizing configurations for different types of parameters including application input options, compiler flags, and runtime/system settings.
- We show that GEIST outperforms expert configuration selection and known sampling approaches based on random selection, Gaussian Process [10], and Canonical Correlation Analysis [13].
- We show that GEIST uses only up to 400 samples for effectively exploring parameter spaces with up to 25,000 configurations.

2 RELATED WORK

Active Harmony is one of the earliest projects aimed at automatic tuning of HPC applications [8, 9]. Since then, a variety of modeling-based methods have been developed for fine-tuning system parameters [11, 29, 31]. At the compiler level, researchers have designed machine learning-based techniques for automatic tuning of the iterative compilation process [25] and tuning of compiler-generated code [24, 28]. Furthermore, several tuning approaches have been developed for application parameter spaces [2, 3]. In general, these approaches target a specific type or subset of parameters, and are often restricted to a component or domain in the HPC or big-data workflow. In contrast, the proposed work does not rely on any domain-specific knowledge, and can take into account the combined influence of different types of parameters.

There also exists a class of autotuners that are designed for multi-objective optimization, examples include RSGDE3 [16], Periscope Tuning Framework [14], and ANGEL [6]. These approaches support only specific types of parameters and certain distributions of the target variable, and operate towards an absolute user-informed objective on the target variable. On the contrary, our approach is designed for handling different types of parameters and distributions, and does not need any form of user-input.

Another important class of methods in this research direction attempt to reduce the resources/time spent in autotuning, through the use of machine learning techniques. Rafiki [22] combines neural networks and genetic algorithms to optimize NoSQL configurations for Cassandra and ScyllaDB. RFHOC [4] uses a random-forest approach to search the Hadoop configuration space. Jamshidi et al. [19] and Roy et al. [26] proposed the use of transfer learning for predicting performance on a target architecture using data collected from another architecture. On the other hand, Grebhahn et al. [15] and Marathe et al. [23] utilized transfer learning to select high-performing combination at a target configuration using domain knowledge extracted from other low-cost configurations. In contrast, our approach relies solely on samples collected for the target problem, and minimizing the number of samples collected is a core objective. Further, our approach avoids the need to build models that perform well for the entire configuration space, and thus needs fewer samples.

The proposed work is most similar to prior efforts that apply statistical machine learning techniques to bootstrap the configuration sampling process [10, 13]. Ganapathi et al. [13] proposed a Kernel Canonical Correlation Analysis (KCCA)- based approach to derive the relationship of parameters with performance and energy. Duplyakin et al. [10] present a Gaussian Process Regression-based method to minimize search space for building regression-based methods in HPC performance analysis. In the paper, we will present a detailed comparison of our approach with these approaches, and show that the proposed approach outperforms these approaches.

3 BOOTSTRAPPING WITH GEIST

The main aim of the proposed work is to identify the best performing configurations for a given application and parameter options.

Although well defined, the space formed by all possible parameters is impractically large in many cases, as a result of which an exhaustive search is infeasible. This section outlines the proposed strategy for smart sampling, which seeks to identify the configurations that result in optimal performance, while observing only a fraction of the entire parameter space.

3.1 Performance Tuning as Adaptive Sampling

Exploring high-dimensional parameter spaces is ubiquitous in different application domains in HPC. One popularly adopted approach for this is to select a subset of samples from the parameter space with the goal of achieving an optimization objective. In our context, a sample corresponds to a specific configuration of system/application-level parameters, while sample collection amounts to actually running the application with a chosen configuration. Most often, the optimization objective is to identify high-performing configurations, if not the best.

The size and complexity of the parameter space can vary significantly across different use cases, thus making it challenging to design a sequential sampling scheme that performs consistently well across use cases. On one extreme, with no prior knowledge about the space, the best one can do is to randomly draw a configuration from the parameter space. On the other extreme, an expert user can make an informed choice based on experience. While the former approach is prone to large variability in the achievable performance, the latter can be limited by the lack of a comprehensive understanding of the interactions between different parameters.

Consequently, in practice, an iterative approach is utilized to progressively obtain samples from regions of high-performance in the parameter space, as determined by a predictive model. Commonly referred to as *adaptive sampling* or *active learning* [27], this approach employs a surrogate model to emulate the process of running the experiment and measuring the performance of a configuration by directly predicting the performance metric. However, such a surrogate model can be plagued by large bias and variance characteristics, arising due to the large range of the metric values, and the lack of a sufficient number of training samples, respectively. Hence, resampling distributions inferred based on the resulting models can be highly misleading.

3.2 Modeling Parameter Spaces using Graphs

In order to address the crucial challenge posed by bias and variance characteristics, we develop a novel bootstrapping approach, called *Good Enough Iterative Sampling for Tuning* (GEIST), for fast tuning of parameters to achieve optimal performance. In GEIST, 1) we represent parameter spaces using undirected graphs, 2) transform the performance metric prediction task into a categorical label prediction task, 3) utilize a state-of-the-art semi-supervised learning technique for label propagation, and 4) perform an iterative sampling pipeline that effectively explores the regions of high-performing parameter configurations. In the rest of this section, we describe this proposed approach.

In contrast to conventional supervised learning approaches, the problem of finding high performing configurations more naturally fits a *transductive* learning framework [20]. In transductive learning, we assume access to the exhaustive set of samples (only configurations, not their performance) in the space that need to be classified, prior to building the model. Given the input set of parameters and their potential values for each application or use case, the exhaustive set of parameter configurations can be easily constructed, thus enabling the use of transductive learning.

Conversely, transductive learning is better suited for the given problem because a broad class of semi-supervised learning methods, which often represent high-dimensional data concisely using neighborhood graphs, fall into this category. The edges in the graph encode the necessary information to perform crucial tasks such as information propagation and data interpolation. Thus, these methods can take advantage of the conventional autotuning wisdom that a high-performing configuration is typically near other high-performing configurations in the parameter space.

Let $G = (V, E)$ denote a undirected graph, where V is the exhaustive set of parameter space configurations ($|V| = N$ nodes), and E is the set of edges, indicating similarity between nodes. In our context, the exhaustive set of parameter configurations $\mathcal{S} = \{\mathbf{x}_i\}_{i=1}^{N}$ is used to construct the neighborhood graph G, where each node is connected to its k nearest neighbors determined based on the Manhattan distance (ℓ_1 norm).

3.3 Reformulating Performance Prediction

As discussed in Section 3.1, using the performance metric as a response variable can lead to models with high bias and variance. Hence, we resort to transforming the continuous performance metric into a categorical variable (*optimal*/*non-optimal*) and employ semi-supervised label propagation to predict the labels at all configurations in $\mathcal{S}$. Given a relatively small, initial sample set $\mathcal{S}_0 = \{\mathbf{x}_i\}_{i=1}^{N_0}$ generated using uniform random sampling, we perform the experiments and build the dataset comprised of the tuples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N_0}$ of size N_0, where y_i denotes the performance metric (e.g. run time or energy) for each case. Without loss of generality, we always define our performance metric in such a way that its value needs to be minimized. Following this, we transform the performance metric for each sample into a categorical label:

$$\mathcal{L}(\mathbf{x}_i) = \begin{cases} \textit{optimal}, & \text{if } y_i \leq \Delta_\ell, \\ \textit{non-optimal}, & \text{otherwise}, \end{cases} \tag{1}$$

where Δ_ℓ denotes the threshold on the performance metric to qualify an experimental run as "optimal". The choice of the hyperparameter Δ_ℓ will be discussed in Section 4.3.1.

3.4 Semi-Supervised Label Propagation

We now describe how the performance labels are propagated using the parameter space graph and training sample set. The problem of propagating labels to nodes in a graph has been well-studied in the machine learning literature under the context of semi-supervised learning [5]. Formally, given a partially labeled graph G, label propagation is aimed at estimating the label probability p_{ik} that a node i is associated with label k. Based on these estimated probabilities, a classification function $C(\mathbf{x}_i) = \arg\max_k p_{ik}$ can then be used to predict the label for that node. In this paper, we utilize Confidence Aware Modulated Label Propagation (CAMLP) [30],

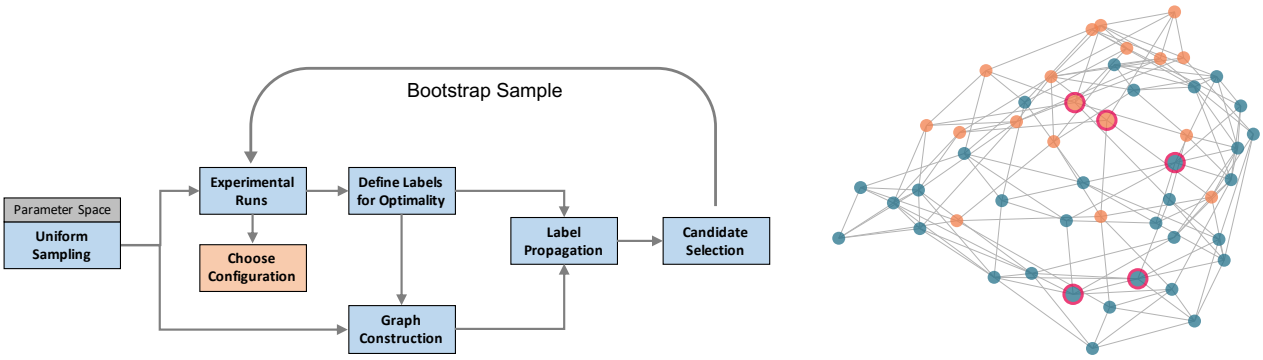


Figure 2: (left) GEIST: Steps for finding high-performing configurations through iterative sampling; (right) Demonstration of the semi-supervised label propagation algorithm used in GEIST. In this example, the large-sized orange and blue nodes correspond to the labeled training samples for *optimal* and *non-optimal* configurations respectively. For the rest of the nodes, we used the *CAMLP* algorithm to propagate the labels, thus predicting the optimality of different configurations in the space.

a state-of-the-art semi-supervised learning algorithm, to perform label propagation.

Broadly speaking, label propagation predicts the labels at unlabeled nodes recursively based on labels of their neighbors. During this process, the predictions are progressively improved until they converge to a stable state. Though a wide variety of strategies exist for propagation, CAMLP achieves improved performance by taking into account both the prior belief at a node and the information propagated from its neighbors during the prediction process. Formally, the label probability at node i for class k is expressed as:

$$p_{ik} = \frac{1}{Z_i}\left(b_{ik} + \beta \sum_{j \in \mathcal{N}(i)} W_{ij} p_{jk}\right). \quad (2)$$

Here, b_{ik} denotes the prior belief on associating node i with label k, $\mathcal{N}(i)$ refers to the set of neighbors of the node i, p_{jk} indicates how strongly a neighboring node j believes that node i has the label k, and W_{ij} is the edge strength between nodes i and j from the adjacency matrix of G. The term β (≥ 0) is referred to as the influence strength parameter, and Z_i is a normalization constant to ensure that p_{ik} sums to 1 for each node i. If a node has a large number of labeled neighbors, it receives a large amount of information from them, thus ignoring the prior belief entirely. In contrast, if a node has only a few labeled neighbors, the prior belief dominates the estimate in Eq. (2).

In summary, CAMLP starts with arbitrary values for p_{ik} and converges to the final predictions by iteratively computing

$$P^t = Z^{-1}\left(B + \beta W P^{t-1}\right), \quad (3)$$

where t and $t-1$ correspond to the current and previous iterations of the label propagation respectively. Note that this is the matrix form of the expression in Eq. (2). Figure 2 (right) demonstrates the working of both the graph construction and label propagation steps. The larger sized nodes indicate the configurations for which we have already collected the data, and the node color indicates its optimality (orange denotes *optimal*). Using the graph structure, the CAMLP algorithm recursively propagates the information and predicts the label at every other unlabeled node in the space (smaller sized nodes). This process has effectively created a distribution in the parameter space that indicates that every orange node has an equally likely chance of being a high-performing configuration, while blue nodes have no evidence of being high-performing. We utilize this labeling scheme to design an iterative algorithm for progressively sampling expected high-performing configurations from $\mathcal{S}$, while avoiding the selection of other configurations.

3.5 GEIST Algorithm

An overview of the proposed iterative scheme that utilizes the techniques described in this section so far is shown in Figure 2 (left) and Algorithm 1. Starting with a uniformly random selection of training samples from the parameter space as the bootstrap set, GEIST uses semi-supervised label propagation to identify potentially optimal candidates from the unseen set. For a random subset of those potentially optimal candidates, experimental results are obtained and the subset is added to the bootstrap set. Next, the steps of semi-supervised label propagation, random subset selection from the potentially optimal candidates, experimental results collection for the subset, and expansion of the bootstrap set using the subset are performed iteratively.

The number of iterations for which GEIST is run can either be determined by the number of experiments that can be executed based on resource availability, or can be based on the configurations obtained in every iteration. For example, if the minimum runtime of configurations obtained so far does not improve in consecutive iterations, the process can be terminated.

Overall, the iterative process of GEIST is trying to explore neighborhoods of high-performing configurations in order to find more high-performing configurations. As such, unlike conventional convex optimization strategies, GEIST does not rely on a single gradient direction to identify the global minimum. Instead, the semi-supervised learning strategy of GEIST can be interpreted as a collection of multiple locally meaningful models, which ends up sampling both local and global minima alike. Intuitively, by progressively sampling in this way, GEIST can better resolve different neighborhoods in the parameter space, and potentially even identify the globally optimal configuration, s_{opt}.

Algorithm 1 GEIST Algorithm

Inputs:
Parameter space $\mathcal{S}$, initial sample size N_0, threshold Δ_ℓ, number of iterations T, number of samples added in each iteration N_+.
procedure
    Initialize bootstrap set $\mathcal{B} = \{\}$.
    Initialize unseen test set $\mathcal{U} = \mathcal{S}$.
    Generate a uniform random sample $\mathcal{S}_0$ of size N_0 from $\mathcal{S}$.
    Update $\mathcal{B} = \mathcal{B} \bigcup \mathcal{S}_0$.
    Construct neighborhood graph G for $\mathcal{S}$.
loop for T iterations:
    Run experiments for samples in $\mathcal{B}$ and build $\{(\mathbf{x}_i, y_i)\}_{i \in \mathcal{B}}$.
    Update $\mathcal{U} = \mathcal{U} \backslash \mathcal{B}$.
    Compute categorical label $\mathcal{L}(\mathbf{x}_i)$, $\forall i \in \mathcal{B}$ using Eq. 1.
    Predict the labels for all configurations in $\mathcal{U}$ using CAMLP.
    Randomly select N_+ *optimal* cases from $\mathcal{U}$ to build $\mathcal{S}_+$.
    Update $\mathcal{B} = \mathcal{B} \bigcup \mathcal{S}_+$.

3.6 Success Metrics

A high-fidelity adaptive sampling strategy is expected to recover most of the optimal configurations while observing the least number of training samples. In a typical scenario, this is measured by the accuracy of the semi-supervised learning approach. However, such an evaluation is not applicable here since we are not interested in recovering the low-performing configurations, and thus are not trying to generate a methodology that predicts well for the entire parameter space. As a result, we adopt the following metrics:

1. Percentile score of Δ_ℓ (PSD-L). This measures how many samples have been added below the initial tolerance threshold Δ_ℓ. A good sampling strategy is expected to add a large number of configurations with performance metric y_i lower than the initial threshold Δ_ℓ and thus lower the cost of sample collection. We measure PSD-L in the bootstrap set $\mathcal{B}$ during every iteration, and expect it to increase in every iteration.

2. Percentile score of Δ_h (PSD-H). Like Δ_ℓ, let us define Δ_h to be the threshold beyond which a configuration is qualified as a low-performing configuration. PSD-H measures how many samples are added above the threshold Δ_h. We expect a good strategy to minimize the inclusion of low-performing configurations, and consequently, we also expect it to increase in every iteration.

3. Best Performing Configuration (BPC). A more straightforward metric is to track the best-performing configuration in the bootstrap set in each iteration of the sampling process. We expect an effective algorithm to identify a high-performing configuration within a few iterations of bootstrapping. In particular, we also expect this best performance to be close to the global optimum in the parameter space, if not the best.

4 EVALUATION SETUP AND DATASETS

In order to evaluate the proposed adaptive sampling approach, GEIST, and compare it with existing approaches, we autotune different types of parameters for optimizing performance metrics such as the execution time and the total energy consumed, of different benchmark applications.

4.1 Benchmarks and Parameter Sources

We use a combination of benchmarks and multiple sources of parameters to create a diverse set of scenarios. In particular, we perform autotuning for compiler flags, application-specific parameters, and runtime options (e.g. OpenMP thread count, power cap).

OpenAtom. OpenAtom [18] is a scalable Charm++-based [1] parallel simulation software for studying atomic, molecular, and condensed phase material systems based on quantum chemical principles. Similar to other Charm++ applications, OpenAtom allows end users to over-decompose the physical domain and the associated work/data units. In order to achieve high performance, it is critical to choose the right level of over-decomposition for different work/data units, and is the subject of our autotuning experiments.

LULESH and compiler flags. LULESH is a shock hydro mini-app developed at Lawrence Livermore National Laboratory. It performs a hydrodynamics stencil calculation using both MPI and OpenMP to achieve parallelism. Among other features, LULESH stresses compiler vectorization, OpenMP overheads, and on node parallelism. Hence, we use LULESH to study and find compiler flags that improve the execution time for single-node runs.

Hypre. Hypre [12] is a parallel linear solver library used in many production applications. It supports many solvers and smoothers, characterized by varying performance and scaling properties. `new_ij` is a test program that allows evaluation of these different options. In this work, we autotune these options and their associated parameters for solving the Laplacian test problem. Laplacian is a 3D Laplace problem discretized using a 27-point finite difference stencil.

Kripke. Kripke is a proxy application for a production transport code for particle physics [21]. In order to enable exploration of novel architectures, it provides several input parameters that change the data structures and code flow, but do not impact the science output. In addition, it can be parallelized using OpenMP. We autotune all these parameters to optimize execution time as well as energy consumption in the presence of a tunable, hardware-enforced power bound.

RAJA policies. RAJA [17] is an abstraction layer for defining looping regions of code that enables developers to easily modify the underlying implementation of different loops without having to rewrite their code. Instead of explicitly writing loops, developers use RAJA to define the body of a loop and its associated "policy", which describes the loop iteration space, the runtime framework for executing it (e.g., sequential or SIMD), and the desired loop iteration order. We autotune parameters of the RAJA loop policies for six different loops in Kripke to optimize overall execution time.

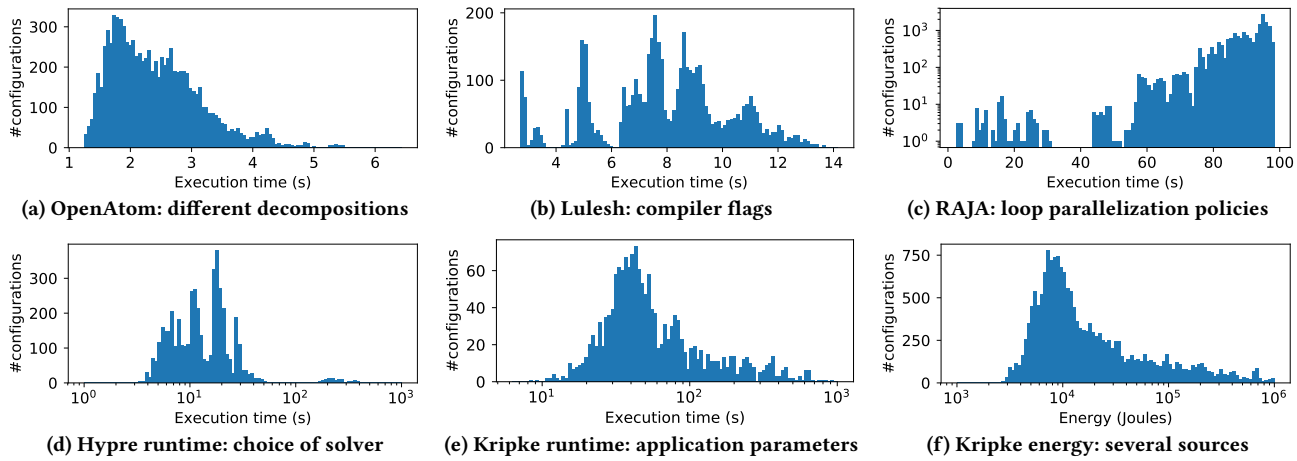
Table 1 summarizes the test cases we use in this paper. Each of these scenarios is discussed in detail in Section 5.

4.2 Distribution of Observed Performance

Figure 3 presents the distribution of the observed performance for different datasets summarized in Table 1. We present these distributions in order to develop familiarity with the search space over which autotuning is being carried out. Note that GEIST, in general and for the results shown in Section 5, does not use any prior knowledge of performance distribution over the search space.

Table 1: Parameter space and performance metric for the use cases explored.

Application	Metric	Parameter type(s)	Parameters	Parameter space
LULESH	Runtime	Compiler flags	-ipo, -fbuiltin, -unroll, -inline-level, -falign-functions etc.	4,800 - 25,920
OpenAtom	Runtime	Decomposition	#chares for electronic states, density, FFT, pair calculation, etc.	8,928
Hypre	Runtime	Solver	solver, smoother, coarsening scheme, interpolation operator	4,580 - 25,198
Kripke	Runtime	Application	nesting order, group set, direction set, #OpenMP threads	1,600
Kripke	Energy	Application, system	power cap and all of above	17,815
RAJA	Runtime	Loop policy	6 loops: sequential, thread-parallel, nested parallelism strategy	18,000



(a) OpenAtom: different decompositions (b) Lulesh: compiler flags (c) RAJA: loop parallelization policies

(d) Hypre runtime: choice of solver (e) Kripke runtime: application parameters (f) Kripke energy: several sources

Figure 3: Examples of distribution of performing metrics to be optimized for various applications due to different sources of parameters. Note the log-scale on the x-axis in the second row due to the large range of the metric.

The evaluation cases that we present in this paper, and other datasets that we have studied, can be broadly divided into three categories. The first category of cases consists of many high-performing configurations. For example, execution times of OpenAtom and LULESH (Figures 3a,3b) over their corresponding parameter spaces exhibit heavily loaded bins on the left. It is interesting to note that, while the performance distribution for OpenAtom shows a single mode at lower execution times, LULESH exhibits a more complex distribution with multiple modes, but still contains strong modes at the bins to the left. For such distributions, it is relatively easy to find a few high-performing configurations because of their abundance.

The second category of cases includes those with few samples close to best performance, followed by bins with higher occupancy, often containing configurations with moderately high performance. Results obtained for Hypre and Kripke (Figures 3d, 3e, 3f) are examples of such distributions (note the log-scale on the x-axis). For such scenarios, while finding a few *good* configurations is easy, identifying the configurations with the highest performance is hard.

The last category is comprised of datasets that are heavily distributed to the right, i.e. they exhibit very few high-performing configurations and most of the configurations provide poor performance. Among our datasets, autotuning of RAJA policies, shown in Figure 3c, is one such scenario. This category is the most challenging in terms of finding high and/or *good* performing configurations.

4.3 Evaluation Methodology

We evaluate the effectiveness of GEIST using the *percentile scores of* Δ_ℓ (PSD-L) and $\Delta_\mathbf{h}$ (PSD-H) and *best-performing configuration* (BPC) metrics described in Section 3.6, and compare against several other approaches (Section 4.4). In order to obtain these metrics, the same input is provided to all methods: a benchmark with a list of parameters and the values each of these parameters can take.

Each method is allowed to query an oracle with a list of configurations (samples) iteratively, for which the oracle provides the experimental value for the metric being optimized. The metric is obtained by conducting a real-world experiment for the given configuration. In our evaluation, for efficiency reasons and for reducing the effect of external factors, we pre-run all configurations and store the information. The oracle simply reads the metric values for the configurations requested by the method from this key-value store. The performance metric values are always stored in a form where lower values are preferred.

4.3.1 Hyper-parameter Selection. All the adaptive sampling methods used in our evaluation, including GEIST, require the selection of four hyper-parameters: size of the initial sample set N_0, the thresholds on the performance metric for classifying a configuration as high-performing Δ_ℓ and low-performing Δ_h, and the number of samples to be added incrementally in each iteration N_+.

In order to ensure statistical stability of the results, N_0 cannot be very small; hence for each dataset and method, we set $N_0 \sim 90$ configurations. For similar reasons, we set $N_+ \sim 50$ for all cases, except

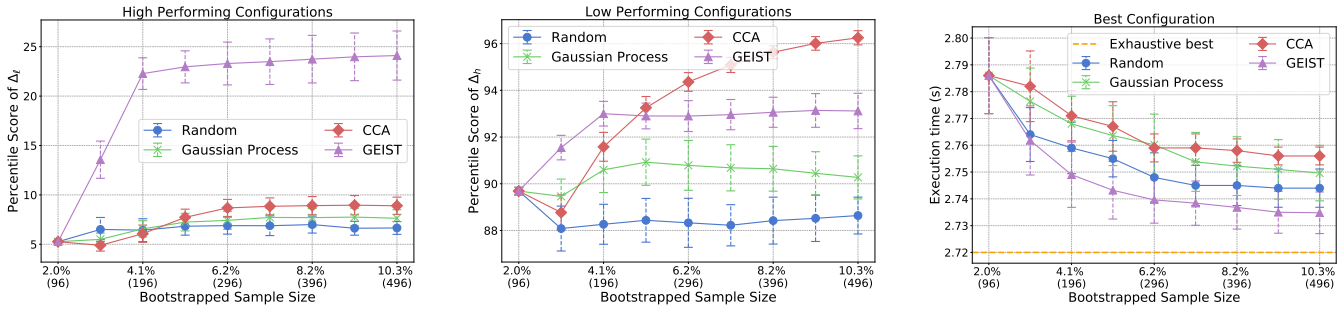


Figure 4: LULESH: GEIST finds 2.6× the number of high-performing configurations in comparison to other methods. CCA is best in avoiding low-performing samples. All methods quickly find configurations close to the global optimum (within 1%).

Kripke for which $N_+ = 16$ because that dataset is relatively small. The choice of Δ_ℓ can depend on the type of application, parameters being tuned, and size of the parameter space. One would prefer to have a very low Δ_ℓ if the parameter space is large, or if one desires to aggressively search for only the very best configuration. However, it is prudent to set Δ_ℓ and N_+ in a way that facilitates the models built for a dataset to provide enough samples for iteratively populating the configuration query list to the oracle. In order to avoid any bias towards a method or from past experience with the benchmarks, we choose Δ_ℓ to be the 5^{th} percentile of the performance metrics from the initial sample set $\mathcal{S}_0$ for all datasets.

The choice of Δ_h does not impact the sampling method and is used for evaluation purpose only. We set it to be the 90^{th} percentile in the initial set, and measure how many extremely slow configurations, and hence experiments, can a method avoid. Finally, the number of iterations, which in practice should be determined by the number of experiments that can be run and the trend in the results obtained, is set to 8 for all methods; we intend to study the trends observed for different datasets and methods across iterations.

4.4 Competing Methods

We now briefly describe the other configuration selection methods that we use for comparison in our experiments.

1. Random Selection: This is the simplest of all sampling strategies, where we add a random set of N_+ samples in each iteration to the bootstrap set. While random sampling is expected to have a large variance, it can be particularly poor at finding good configurations using only a limited number of samples.

2. Gaussian Process-based Adaptive Sampling: This is a common sampling technique in UQ (Uncertainty Quantification) applications, where the samples to be added to the training set are chosen based on both the expected metric value and the prediction uncertainty from a Gaussian Process regressor. The intuition here is that predictions with a large variance lie in regions of high uncertainty. Hence, in each iteration, we add samples that are predicted to be high performing, as well as the ones with large variance, to improve the model in the subsequent iterations.

3. CCA based Neighborhood Selection: Similar to the approach in [13], we utilize canonical correlation analysis to learn a mapping $\mathbf{V}$ such that $\mathbf{V}^T\mathbf{X}$ is maximally correlated with the performance metric $\mathbf{y}$, using the samples in the bootstrap set. In each iteration, we choose N_+ nearest neighbors to the current best configuration and add them to the bootstrap set.

4. Expert Choice: We include performance against a manually determined near-optimal configuration by an expert practitioner.

5. Exhaustive Search (Oracle): In order to get a sense of how well we are able to find the optimal configuration(s), we also compare our method against the best performance that can be obtained on an application, that is found using an exhaustive search.

5 EVALUATION

In this section, we evaluate and compare GEIST with other methods described in Section 4.4 on the benchmark datasets in Table 1. For each dataset, we perform 50 adaptive sampling experiments for every method, and report the observed mean and standard deviation for each of the metrics. For all methods and data sets, the same set of 50 random seeds was used for generating the initial sample sets.

5.1 Compiler Flags for LULESH

Users often rely on the default choice of flags enabled by the *-O3* flag to obtain the best performance that can be provided by a compiler. However, it has been shown that the default options enabled by *-O3* may not be best-suited for every application, and performance can be gained by tuning the individual flags [7].

We autotune the compiler flags for LULESH as our first use case. Because we want to compare the best-performing configuration obtained by various methods with the *exhaustive best*, we limit our exploration to 9-10 compiler flags, so that exhaustive collection of data is possible. Some of the flags used are listed in Table 1. The runtime obtained with the *-O3* flag is 6.02 seconds.

Figure 4 compares the results obtained for autotuning using GEIST and other competing methods. The initial sample size for these experiments was 96, and 50 samples were added in every iteration. We observe that GEIST finds significantly more (~ 2.6×) high-performing configurations in comparison to other methods. GEIST also outperforms random selection and Gaussian Process based sampling in avoiding low-performing configurations, but CCA outperforms GEIST in that metric. All methods quickly find configurations close to the global optimum, which is not far away from the best configuration in the initial random sample set. This result can be explained by the distribution presented in Figure 3b, which shows that several high-performing configurations exist.

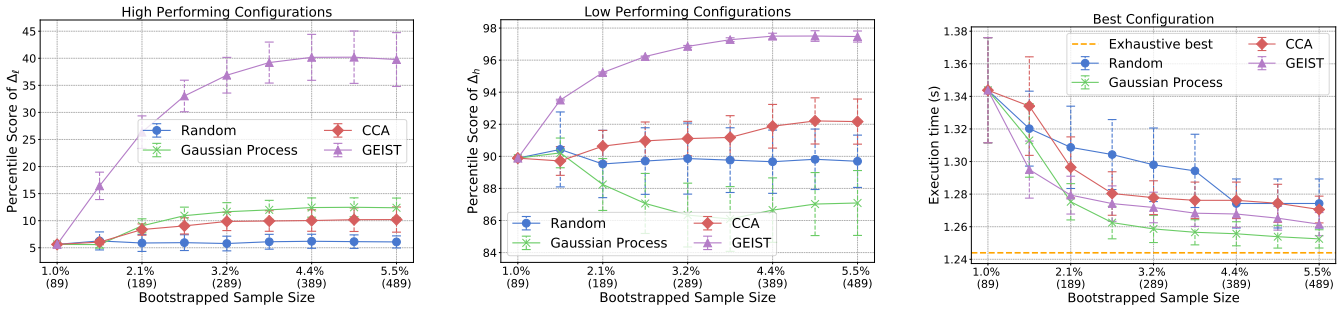


Figure 5: OpenAtom: GEIST discovers significantly higher number of high-performing configurations, and avoids low-performing configurations in comparison to other methods. GEIST and Gaussian Process are able to find configurations that perform close to the optimum (within 3% of the global best) using only 189 observations (90 initial + 50 each in 2 iterations).

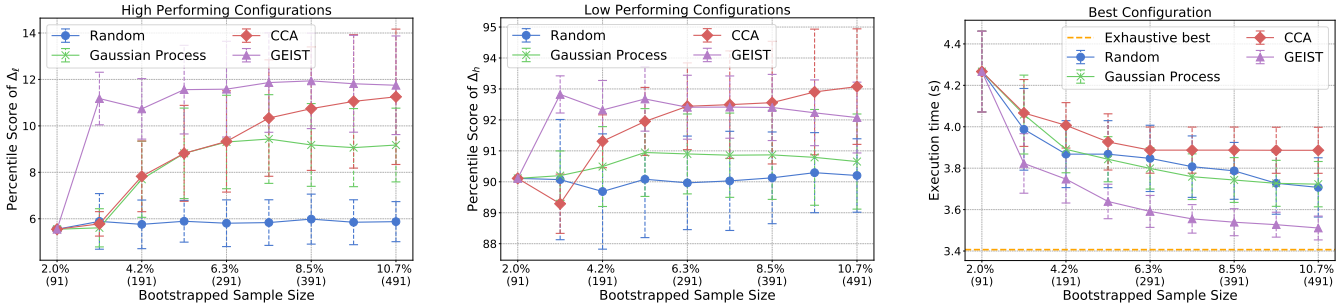


Figure 6: Hypre: GEIST finds near-optimal configuration using only 341 samples (91 initial + 50 each in 5 iterations). These configurations are 5.6% and 9% faster than those found by Gaussian Process and CCA, respectively, using 341 samples.

Nonetheless, the best-performing configuration obtained from all methods is significantly (2.2×) faster than the typical default of *-O3*.

We also performed similar experiments with three other sets of compiler flags for parameter space sizes up to 25,920. For all scenarios, we obtained data distributions and autotuning results similar to those presented above. However, the global best performance obtained heavily depends on the compiler flags being explored and ranges from 2.72s to 5.92s. Nonetheless, all methods are able to find configurations that perform close to the optimum, and GEIST finds significantly more high-performing configurations.

5.2 Decomposition Selection for OpenAtom

In OpenAtom, users can decompose different tasks into different numbers of work units. This flexibility leads to a large parameter space, in which each configuration can take several minutes to execute. For the science problem simulated in this paper (32 molecules of Water on 128 nodes of a Blue Gene/Q [18]), an expert user would choose a configuration that takes 1.6 seconds per step.

Figure 5 shows that, similar to LULESH, GEIST identifies significantly higher (4×) number of high-performing configurations in comparison to the other methods. Unlike other methods, GEIST also successfully avoids exploring low-performing configurations. However, like LULESH, the dataset of OpenAtom tested by us contains many high-performing configurations (Figure 3a) and hence most methods are able to quickly find near-optimal (within 3% of the global best of 1.24 s) configurations in 2 to 3 iterations of adaptive sampling. Gaussian Process based sampling and GEIST requires the minimum number of samples (189) to find these configurations, while random selection performs the worst and needs 389 samples.

5.3 Solver Selection for Hypre

The `new_ij` benchmark of the hypre suite allows the use of four parameters: solver, smoother, coarsening scheme, and interpolation operator, which can create a parameter space of size 4, 580. By also modifying the power bounds, this parameter space increases to up to 25, 198. We autotuned parameters with and without including different power bounds, and achieved similar results for both, so henceforth we discuss the results without power bounds only.

Figure 6 shows that, except random selection, all other methods are able to find many high-performing configurations. However, only GEIST is able to iteratively improve the performance of configurations found, thus determining configurations within 3% of the global best. These configurations found by GEIST are 5.6% and 9% better than the best configurations found by the next best methods, Gaussian Process and CCA, respectively. Moreover, it only takes 341 samples for GEIST to find the near-optimal configurations.

GEIST is able to outperform other methods for hypre because it is able to identify the very few high-performing configurations that are in the left-most bins of Figure 3d. While other methods are able to only find the good configurations from heavily occupied bins, GEIST is able effectively to explore the neighborhoods of those configurations and find the near-optimal configurations.

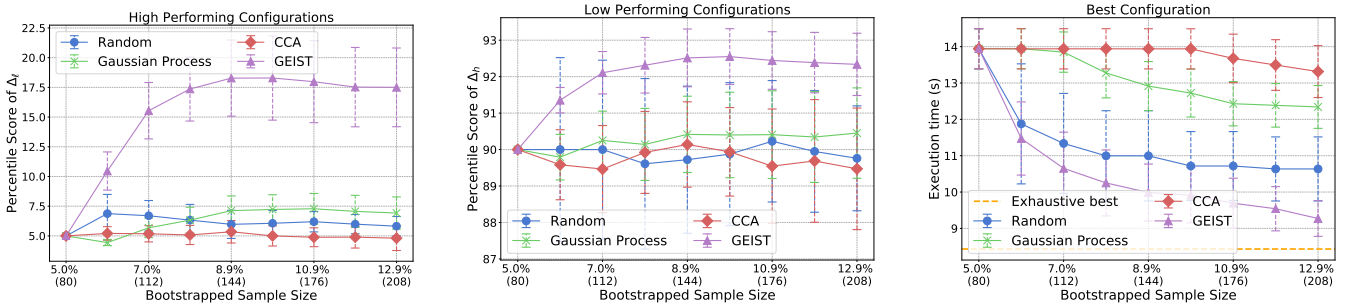


Figure 7: Kripke time: GEIST outperforms all other methods and finds configurations that are within 19% and 10% of global best using 144 and 208 samples. The next best method is random selection which is 30% and 26% slower than the global best for these sample counts. Note that due to the small size of this dataset, only 16 samples are added in each iteration.

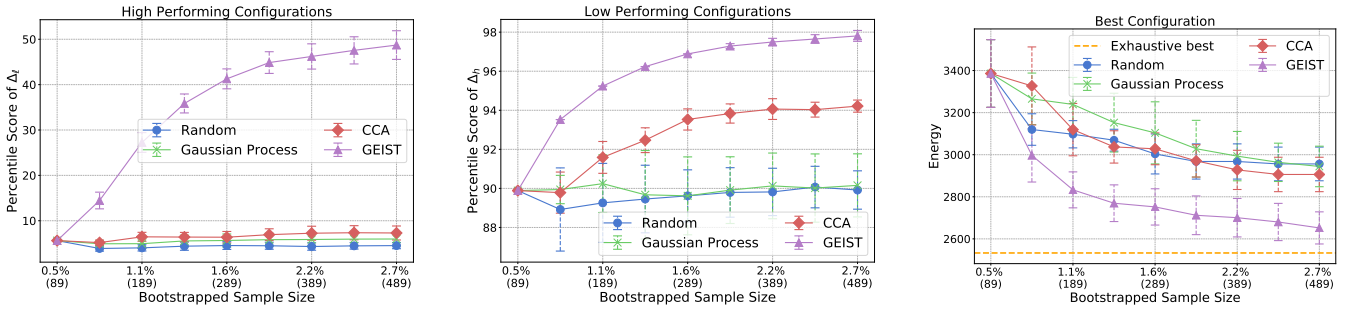


Figure 8: Kripke energy: GEIST is significantly better at finding low-energy configurations, avoiding very high-energy configurations, and finds configurations that consume ∼ 9% lower energy than configurations found by other methods.

5.4 Kripke: Time and Energy Optimization

In order to explore different architectural features and provide performance portability, Kripke provides several application-level options to change the code control flow without changing the science performed. Table 1 list these options: different orderings for executing compute kernels, number of group and energy sets to overlap computation and communication, and the OpenMP thread count. We explore this space to find configurations with minimum runtime. Additionally, by enabling power capping, we also search for configurations that minimize total energy consumption of the execution. An expert user's choice in this benchmark would have been to manually test for each loop ordering with a few group/energy sets, and optimize for energy at 2^{nd}-3^{rd} highest power level. This would have resulted in the execution time of 15.2 seconds and energy consumption of 4,742 Joules.

Figure 7 shows that GEIST outperforms all other methods comprehensively in finding configurations with low execution time, and is also better at avoiding configurations with high execution time. GEIST finds configurations that are within 19% and 10% of the globally optimal configuration of 8.43s using only 144 and 208 samples, respectively. These runtimes are significantly better than the runtimes obtained using random selection (27%), Gaussian Process (48%), and CCA (59%) methods, with a total sample size of 208.

Similar results are obtained for optimizing energy consumption, as shown in Figure 8. GEIST finds significantly higher numbers of low-energy configurations (6×) and is also the best method for avoiding high-energy configurations. For any given iteration or sample count, GEIST finds best configurations that consume ∼ 9% lesser energy than best configurations discovered by other methods. The best configuration found by GEIST is within 4% of the global optimum of 2,533 Joules and needs only 2.5%(339) samples of the total parameter space.

Like hypre, we believe that GEIST is able to improve upon other methods for finding the best-performing configurations because of the distribution of Kripke datasets (Figures 3e and 3f). GEIST uses the parameter graph neighborhood relations to explore the neighborhoods of high-performing configurations and find the few near-optimal configurations in the left-most bins.

5.5 Selecting RAJA policies

Six different RAJA loops were used in our benchmark, five of which are nested loops with three to five nesting levels. The underlying loop policies for each of these loops can be chosen at runtime, and includes options to execute sequentially or with thread parallelism and to select the nesting level to invoke a parallel OpenMP region. Since different loop policies populate processor caches differently, we cannot tune loops independently and must explore the combined space of all policies and loops. An expert user would use OpenMP at the outermost level and obtain 57.2s runtime.

Figure 9 compares the quality of configurations discovered by GEIST with other methods. With increasing iteration count and samples, we find that GEIST progressively gets better at selecting high-performing configurations while all other methods exhibit

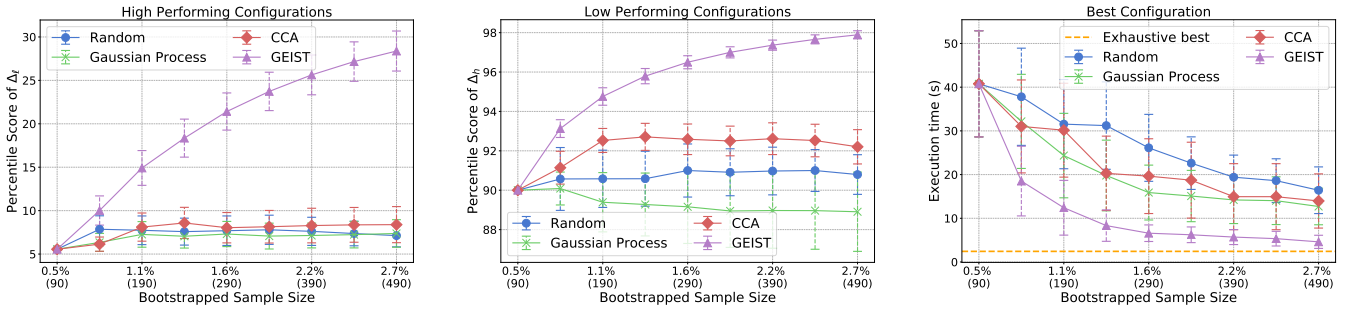


Figure 9: RAJA policy: For this heavily skewed dataset, GEIST is the only method that identifies configurations close to the global optimum. Configurations obtained using GEIST are 2.4× and 2× slower than the global optimum using 290 and 490 samples respectively, while the second best method (Gaussian Process) finds configurations that are 6.5× and 5.22× slower.

Table 2: Results summary. Units: runtime - seconds, energy - Joules. Collection cost includes compilation and runtime.

Application/ Metric	Parameter space size (collection cost)	Exhaustive best perf.	Expert best perf	Competition best perf (%high confs)	GEIST best perf (%high confs)	#samples used (collection cost)
Lulesh/Runtime	4,800 (19.9 hrs)	2.72	6.02 (-O3)	CCA - 2.74 (9)	2.73 (24)	246 (1.4 hrs)
OpenAtom/Runtime	8,928 (111.6 hrs)	1.24	1.6 (symmetric decomposition)	GP - 1.25 (10)	1.26 (39)	189 (2.4 hrs)
Hypre/Runtime	4,580 (24.9 hrs)	3.40	Unknown	GP - 3.70 (9)	3.51 (11)	341 (1.6 hrs)
Kripke/Runtime	1,600 (38.9 hrs)	8.43	15.2 (few sets and threads)	Rand - 10.6 (5)	9.27 (17)	208 (4 hrs)
Kripke/Energy	17,815 (321K J)	2533	4742 (2^{nd}-3^{rd} highest power)	CCA - 2906 (7)	2652 (43)	339 (1836 J)
RAJA/Runtime	18,000 (444 hrs)	2.43	57.28 (all OpenMP)	GP - 12.6 (7)	4.61 (28)	390 (8.6 hrs)

marginal improvement. Similar trends are observed for selection of low-performing configurations, wherein GEIST progressively gets better at avoiding low-performing configurations.

Figure 9 also shows that the best configurations discovered by GEIST are ∼ 2.7× faster than the best configurations found using other methods. GEIST produces configurations that are 2.4× and 2× slower than the global optimal of 2.43s using only 290 and 490 samples respectively. In contrast, the second best method (Gaussian Process) can only identify configurations that are 6.5× and 5.22× slower for these sample counts. These results highlight that when the distribution is heavily skewed to the left (Figure 3c), GEIST is significantly better than known methods in finding high-performing neighborhoods and best configurations within those neighborhoods. In summary, regardless of the inherent distribution of the performance metric in their corresponding parameter spaces, GEIST produces near-optimal configurations for all benchmarks while consistently outperforming all competing methods.

6 DISCUSSION AND CONCLUSION

Table 2 summarizes the evaluation results presented in this paper. Broadly speaking, we see that for all test cases, GEIST is able to find high-performing configurations that are closer to the global optimum with fewer samples in comparison to other methods. The method which is second best to GEIST varies with the dataset being tuned. Furthermore, because GEIST quickly finds more high-performing configurations than other methods, each training iteration becomes progressively cheaper to sample than the previous, thus speeding up the process towards convergence.

An in-depth look at the optimal configurations selected revealed that often times, the configurations that provide the best performance are not intuitive, nor are they well-known to expert users. For example, in OpenAtom, the expert users tend to pick symmetric decompositions for multi-dimensional physical entities. However, significantly better performance is obtained using asymmetric decompositions (1.6s vs 1.26s). Similarly, for RAJA policies, experienced users expect OpenMP loop at outermost levels to work well, but we find that a complex combination of loop levels provides significantly better performance (57.28s vs 4.61s). Nonetheless, despite being unaware of the domain or parameter types, GEIST is able to find high-performing configurations after few sampling iterations.

Finally, our study suggests that the difference between high-performing configurations chosen by GEIST and other methods increases as the distributions of performance metrics move to the right; i.e., when fewer high-performing configurations are available, GEIST is able to find them, but other methods do not. This is inherent in the design of GEIST, which uses sampling to intelligently avoid large parameter spaces with under-performing samples.

In conclusion, we have presented and shown that an adaptive sampling strategy that is able to exploit neighborhood relationships among configurations in the parameter space is very good at finding near-optimal configurations with few samples. We hope that this scheme, which does not require information about the domain, metric distribution, or user input, will help the HPC community autotune its codes using minimal resources.

REFERENCES

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable

Objects: Charm++ in Practice *(SC)*.
[2] Prasanna Balaprakash, Robert B Gramacy, and Stefan M Wild. 2013. Active-learning-based surrogate models for empirical performance tuning. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 1–8.
[3] David Beckingsale, Olga Pearce, Ignacio Laguna, and Todd Gamblin. 2017. Apollo: Reusable models for fast, dynamic tuning of input-dependent code. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE.
[4] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng. 2016. RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (May 2016), 1470–1483. https://doi.org/10.1109/TPDS.2015.2449299
[5] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. 2009. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks* 20, 3 (2009), 542–542.
[6] Ray S Chen and Jeffrey K Hollingsworth. 2015. Angel: A hierarchical approach to multi-objective online auto-tuning. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 4.
[7] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. 2010. Evaluating Iterative Optimization Across 1000 Datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 448–459. https://doi.org/10.1145/1806596.1806647
[8] I-H Chung and Jeffrey K Hollingsworth. 2006. A case study using automatic performance tuning for large-scale scientific programs. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*. IEEE, 45–56.
[9] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*. IEEE Computer Society Press.
[10] Dmitry Duplyakin, Jed Brown, and Robert Ricci. 2016. Active Learning in Performance Analysis. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*. IEEE, 182–191.
[11] Thomas L Falch and Anne C Elster. 2017. Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications. *Concurrency and Computation: Practice and Experience* 29, 8 (2017).
[12] R.D. Falgout, J.E. Jones, and U.M. Yang. 2006. The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, A.M. Bruaset and A. Tveito (Eds.). Vol. 51. Springer-Verlag, 267–294.
[13] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. 2009. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX conference on Hot topics in parallelism*. USENIX Association.
[14] Michael Gerndt and Michael Ott. 2010. Automatic performance analysis with periscope. *Concurrency and Computation: Practice and Experience* 22, 6 (2010).
[15] Alexander Grebhahn, Norbert Siegmund, Harald Köstler, and Sven Apel. 2016. Performance prediction of multigrid-solver configurations. In *Software for Exascale Computing*. Springer, 69–88.
[16] Philipp Gschwandtner, Juan José Durillo, and Thomas Fahringer. 2014. Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage.. In *Euro-Par*. 87–98.
[17] R D Hornung and J A Keasler. 2014. *The RAJA Portability Layer: Overview and Status*. Technical Report LLNL-TR-661403. Lawrence Livermore National Laboratory.
[18] Nikhil Jain, Eric Bohm, Eric Mikida, Subhasish Mandal, Minjung Kim, Prateek Jindal, Qi Li, Sohrab Ismail-Beigi, Glenn Martyna, and Laxmikant Kale. 2016. OpenAtom: Scalable Ab-Initio Molecular Dynamics with Diverse Capabilities. In *International Supercomputing Conference (ISC HPC '16)*.
[19] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 497–508.
[20] Thorsten Joachims. 2003. Transductive learning via spectral graph partitioning. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*.
[21] AJ Kunen, TS Bailey, and PN Brown. 2015. KRIPKE-A massively parallel transport mini-app. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep* (2015).
[22] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. 2017. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/3135974.3135991
[23] Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. 2017. Performance Modeling under Resource Constraints Using Deep Transfer Learning. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. IEEE Computer Society. LLNL-CONF-736726.
[24] Saurav Muralidharan, Manu Shantharam, Mary Hall, Michael Garland, and Bryan Catanzaro. 2014. Nitro: A framework for adaptive code variant tuning. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE.
[25] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Minimizing the cost of iterative compilation with active learning. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 245–256.
[26] Amit Roy, Prasanna Balaprakash, Paul D Hovland, and Stefan M Wild. 2016. Exploiting performance portability in search algorithms for autotuning. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE.
[27] Burr Settles. 2012. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 6, 1 (2012), 1–114.
[28] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–12.
[29] Ananta Tiwari and Jeffrey K Hollingsworth. 2011. Online adaptive code generation and tuning. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 879–892.
[30] Yuto Yamaguchi, Christos Faloutsos, and Hiroyuki Kitagawa. 2016. Camlp: Confidence-aware modulated label propagation. In *Proceedings of the 2016 SIAM International Conference on Data Mining*. SIAM, 513–521.
[31] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. *SIGPLAN Not.* 51, 4 (2016), 545–559.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-750296).