Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization

Harshitha Menon*, Abhinav Bhatele†, Todd Gamblin*

*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551 USA
†Department of Computer Science, University of Maryland, College Park, Maryland 20742 USA
E-mail: *{harshitha, tgamblin}@llnl.gov, †bhatele@cs.umd.edu

Abstract—**High performance computing applications, runtimes, and platforms are becoming more configurable to enable applications to obtain better performance. As a result, users are increasingly presented with a multitude of options to configure application-specific as well as platform-level parameters. The combined effect of different parameter choices on application performance is difficult to predict, and an exhaustive evaluation of this combinatorial parameter space is practically infeasible. One approach to parameter selection is a user-guided exploration of a part of the space. However, such an ad hoc exploration of the parameter space can result in suboptimal choices. Therefore, an automatic approach that can efficiently explore the parameter space is needed. In this paper, we propose HiPerBOt, a Bayesian optimization based configuration selection framework to identify application and platform-level parameters that result in high performing configurations. We demonstrate the effectiveness of HiPerBOt in tuning parameters that include compiler flags, runtime settings, and application-level options for several parallel codes, including, Kripke, Hypre, LULESH, and OpenAtom.**

Keywords-**parameter selection, auto-tuning, transfer learning, performance modeling, bayesian optimization**

I. Introduction

High performance computing (HPC) applications, runtimes, and platforms have become highly configurable. The high configurability enables applications to extract maximal performance. Unfortunately, for users, the same configurability has led to an increasingly large combinatorial search space of application-specific as well as platform-level parameters. These configurable parameters can be compiler flags, runtime system settings, application-specific parameters, hardware-level options etc. A particular choice of values for all the parameters, often called a configuration, can significantly impact various application performance metrics, such as execution time, power consumption, and system throughput among other things.

Finding the best configuration in the case of a large number of parameters is challenging because it involves exploring the combinatorial parameter configuration space. While ad hoc manual tuning by the user may yield satisfactory results, it usually requires the user to have expert domain knowledge to guide the tuning process. Moreover, in the case of a naïve user, manual tuning can result in a suboptimal configuration. Hence, automatic performance tuning techniques have become necessary to identify high-performing configurations. Autotuning can either exhaustively evaluate all parameter choices or perform directed search of the parameter space. However, exhaustive methods work only for very small number of parameters as a large number of parameters makes the parameter space combinatorially large and infeasible to explore exhaustively. Therefore, model-based methods are used for autotuning, as they can replace expensive empirical evaluations.

Several supervised as well as semi-supervised learning based models have been applied to select high-performing configurations [1]–[3]. For these methods to perform well, they require a large number of labeled instances that sufficiently cover the search space. Moreover, the training instances are randomly selected, irrespective of their usefulness in achieving the true goal of the task. Note that the task of selecting high-performing configurations does not require models to perform well over the entire configuration space, which gives us the opportunity to use fewer but informative instances to learn a model that focuses on high performing configurations.

In this paper, we present HiPerBOt, an active learning framework based on Bayesian optimization [4], to identify application and platform-level parameters that result in high-performing configurations of HPC applications. Active learning allows the learning algorithm to choose the data from which it learns the most to achieve a specified objective. This approach is particularly beneficial when evaluating the objective function, which in this case is running the application, is very expensive. We designed HiPerBOt to allow easy transferability of learning from a low-cost representative source data to target data. This is especially useful in the HPC domain, where codes are tuned for large scale based on a performance tuning study at small scale and a few experiments at large scale. HiPerBOt will enable users to select good configurations for their applications using limited evaluations, reducing the user effort and resource overhead.

Our main contributions in this work are the following:

- Application of Bayesian optimization in the context of HPC application performance to tune parameters that include compiler flags, runtime settings, and application-level options.
- HiPerBOt, an active learning framework based on the bayesian optimization method specifically catering to the needs of HPC application tuning.
- Extending HiPerBOt to seamlessly support transfer learning, which involves transferring the learning from a low-cost source data to a target data, and use it for performance tuning when limited resources are available for collecting data at scale for a target application.

- Using the model learnt by HiPerBOt to analyze the relative importance of different parameters in selecting high-performing configuration.
- A broad experimental evaluation of HiPerBOt and comparison with recently published state-of-the-art approaches on a range of HPC applications and mini-applications, including, Kripke, HYPRE, LULESH, and OpenAtom.

In our evaluation, we find that HiPerBOt performs significantly better than competitive methods for selecting the best-performing configuration while using limited number of evaluations. In section IV we show that HiPerBOt uses 50% fewer evaluations to find the best configuration for Kripke in comparison to a competitive method.

II. Background on Bayesian Optimization

Optimization problems involve minimizing an objective function $f(\boldsymbol{x})$, and in many cases this objective is a black box function. Evaluating this function amounts to sampling at a point x and observing the output value. When the objective function is expensive to evaluate, e.g. running large scale application runs for parameter tuning, it is important to minimize the number of samples evaluated. This is where Bayesian optimization [4] technique is very powerful. Bayesian optimization has been used in the hyperparameter tuning for Machine Learning, especially in tuning the hyperparameters of deep neural networks [5], where training a neural network is very expensive. Bayesian optimization approximates the original, expensive to evaluate, objective function with a surrogate objective, which is significantly cheaper to compute. In the following, we describe the construction of the surrogate objective function.

A. Surrogate Objective Construction

The surrogate objective $\mathcal{I}(\boldsymbol{x})$ is a model of the true objective function $f(\boldsymbol{x})$. It is significantly cheaper to compute and thus can be computed for a very large set of possible values of x. From this set, the most promising values can then be selected to evaluate the more expensive true objective $f(\boldsymbol{x})$. Typically, a probabilistic surrogate model $p_{y|\boldsymbol{x}}(y \mid \boldsymbol{x})$ is used. The surrogate model defines a probability density over the objective values y given $\boldsymbol{x}$. A good surrogate model would assign a high probability to the correct value $f(\boldsymbol{x})$. Given the surrogate model $p_{y|\boldsymbol{x}}(y \mid \boldsymbol{x})$ and some $\boldsymbol{x}$, Expected Improvement ($\mathcal{I}$) [6] is the expected margin by which the true objective $f(\boldsymbol{x})$ will perform better than a threshold value $y^{(\tau)}$ of the objective function:

$$\mathcal{I}(\boldsymbol{x}, y^{(\tau)}) = \int_{-\infty}^{y^{(\tau)}} (y^{(\tau)} - y) p_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}) dy \qquad (1)$$

For an objective value y, the margin of improvement over the threshold $y^{(\tau)}$ is defined as $\max\{y^{(\tau)} - y, 0\}$. It is non-zero only for those values of objective that are smaller than the threshold $y^{(\tau)}$ and thus lead to an improvement. In many cases $y^{(\tau)}$ is the best-observed value so far. However, often a more conservative value is used. This margin captures the intuition that a given $\boldsymbol{x}$ is useful only if it provides an improvement with respect to the best currently known performance value.

The surrogate model $p_{y|\boldsymbol{x}}(y \mid \boldsymbol{x})$ is used to predict the likely values of the objective for a given $\boldsymbol{x}$ without conducting the expensive evaluation $f(\boldsymbol{x})$. In Bayesian optimization method, instead of using $p(y \mid x)$, Bayes rule is used to define $p_{y|\boldsymbol{x}}(y \mid \boldsymbol{x})$ in terms of $p_{\boldsymbol{x}|y}(\boldsymbol{x} \mid y)$, yielding:

$$p_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}) = \frac{p_{\boldsymbol{x}|y}(\boldsymbol{x} \mid y) p_y(y)}{p_{\boldsymbol{x}}(\boldsymbol{x})} \qquad (2)$$

Here, $p_y(y)$ is the prior distribution for y and $p_{\boldsymbol{x}}(\boldsymbol{x}) = \int p_{\boldsymbol{x}|y}(\boldsymbol{x} \mid y) p_y(y) dy$ is the marginal distribution of $\boldsymbol{x}$.

Modeling $p_{\boldsymbol{x}|y}(\boldsymbol{x} \mid y)$ is still infeasible due to insufficient data. However, note that in each iteration we are interested in evaluating whether a configuration would perform better than the best found so far. This observation is used to binarize the space of y into two possibilities: 1) performance better than some threshold $y^{(\tau)}$ i.e. $y < y^{(\tau)}$, and 2) performance worse than some threshold $y^{(\tau)}$ i.e. $y \geq y^{(\tau)}$. Here, $y^{(\tau)}$ is used as a measure of the best performance so far.

Hence, we define the probability distribution $p_{\boldsymbol{x}|y}(\boldsymbol{x} \mid y)$ in terms of two probability density functions; $p_g(\boldsymbol{x})$ for good performance ($y < y^{(\tau)}$), and $p_b(\boldsymbol{x})$ for bad performance ($y \geq y^{(\tau)}$):

$$p_{\boldsymbol{x}|y}(\boldsymbol{x} \mid y) = \begin{cases} p_g(\boldsymbol{x}) & \text{if } y < y^{(\tau)} \\ p_b(\boldsymbol{x}) & \text{if } y \geq y^{(\tau)}, \end{cases} \qquad (3)$$

As we will see below, the above definition allows us to compute the surrogate objective up to a scaling factor.

One option is to choose $y^{(\tau)}$ as the best-observed value of the objective so far. However, this may not be robust and may exhibit large variance leading to problems in estimating $p_{\boldsymbol{x}|y}(\boldsymbol{x} \mid y)$. Instead, $y^{(\tau)}$ is defined in terms of α-quantile for stability i.e. $y^{(\tau)}$ is chosen such that $p(y < y^{(\tau)}) = \alpha$. With this choice and using eq. (3), we can write $p_{\boldsymbol{x}}(\boldsymbol{x})$ as

$$p_{\boldsymbol{x}}(\boldsymbol{x}) = p_g(\boldsymbol{x})\alpha + p_b(\boldsymbol{x})(1 - \alpha) \qquad (4)$$

Substituting eq. (2) in eq. (1) and using eq. (4), we get:

$$\mathcal{I}(\boldsymbol{x}, y^{(\tau)}) = \frac{1}{\alpha + \frac{p_b(\boldsymbol{x})}{p_g(\boldsymbol{x})}(1 - \alpha)} \qquad (5)$$

From above we see that the expected improvement $\mathcal{I}(\boldsymbol{x}, y^{(\tau)})$ is greatest for those values of $\boldsymbol{x}$ whose $\frac{p_g(\boldsymbol{x})}{p_b(\boldsymbol{x})}$ ratio is highest. Therefore, instead of computing the value in eq. (1) exactly, we can just compute the ratio $\frac{p_g(\boldsymbol{x})}{p_b(\boldsymbol{x})}$ for any candidate $\boldsymbol{x}$. With this observation, to compare a set of candidate configurations, we can compute the ratio $\frac{p_g(\boldsymbol{x})}{p_b(\boldsymbol{x})}$ for each candidate and return the one with the highest value of the ratio as the selected candidate $\boldsymbol{x}_t^*$. $\boldsymbol{x}_t^*$ can then be used for performing the full evaluation $y_t^* = f(\boldsymbol{x}_t^*)$.

III. The HiPerBOt Framework

Our framework uses Bayesian optimization based active learning method to identify top performing configurations (maximize certain application dependent metric) while minimizing the number of full application runs. The technique we use builds upon the approach described by Bergstra et al. [5] where they apply it for hyperparameter optimization for Machine Learning. Since computing the value of the objective for a configuration by running the full application is expensive, we construct a surrogate model using Bayesian optimization described in section II. The surrogate model is cheap to compute and can be used to efficiently evaluate a large number of configurations to select a suitable candidate. Below, we first set up the problem and provide an overview of the iterative method used in our framework. Next, we state the full algorithm and describe its application to transfer learning.

Let the application consist of n tunable parameters, each of which is represented by $x_i, i \in \{1, \ldots, n\}$. We use $\boldsymbol{x} = [x_1, \ldots, x_n]$ to represent a configuration of the n parameters as a vector. Further, let $f(\boldsymbol{x})$ represent the performance of a particular configuration $\boldsymbol{x}$ computed by running the full application. Our goal is to find an optimal parameter configuration $\boldsymbol{x}^*$ that produces the best application performance $f(\boldsymbol{x}^*)$. This can be written down as the following objective:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{6}$$

However, directly optimizing the above objective is not feasible since $f(\boldsymbol{x})$ is expensive to compute. Therefore, we use an iterative model based method that optimizes a sequence of surrogate objectives $\mathcal{I}_t(\boldsymbol{x}), t \in \{1, \ldots, T\}$ that are efficient to compute instead of the expensive true objective $f(\boldsymbol{x})$. With increasing iterations, the model used by our framework more accurately models $f(\boldsymbol{x})$, leading to better and better parameter configurations. This method falls in the class of Sequential Model Based Global-Optimization methods (SMBO) that have previously been studied for black box optimization of functions that are expensive to evaluate [6]–[9].

A. Overview of the Iterative Algorithm

Our framework uses an iterative model based algorithm. Let t index the iterations up to the maximum iteration T. In each iteration a single configuration $\boldsymbol{x}_t^*$ is chosen to compute the value of the true objective as $y_t^* = f(\boldsymbol{x}_t^*)$. The pair $(\boldsymbol{x}_t^*, y_t^*)$ is then added to the observation history to produce $\mathcal{H}_t$. The observation history $\mathcal{H}_t$ is the set of all the computed objectives and is updated as $\mathcal{H}_t = \mathcal{H}_{t-1} \cup \{(\boldsymbol{x}_t^*, y_t^*)\}$. $\mathcal{H}_t$ is then used to help choose $\boldsymbol{x}_{t+1}^*$ in the next iteration.

To choose $\boldsymbol{x}_t^*$ in iteration t, a set of candidate configurations are evaluated. Since the objective $f(\boldsymbol{x})$ is expensive to compute, evaluating a lot of candidates is not feasible. Using the observation history $\mathcal{H}_{t-1}$, a cheap to evaluate model surrogate $\mathcal{I}_t(\boldsymbol{x})$ is constructed. The parameter configuration $\boldsymbol{x}_t^*$ that optimizes the surrogate $\mathcal{I}_t(\boldsymbol{x})$ at time t is then treated as the best candidate to evaluate $f(\boldsymbol{x})$ at time t. The algorithm starts with a small initial set of parameter configurations obtained by randomly sampling the parameter space. All of these initial configurations are evaluated using $f(\boldsymbol{x})$ to produce the initial observation history $\mathcal{H}_0$. $\mathcal{H}_0$ is then used to produce the initial model iterate $\mathcal{I}_1(\boldsymbol{x})$ and the algorithm continues iteratively until iteration $t = T$.

B. Probability Density Function for Configuration Space

Recall that $\boldsymbol{x} = [x_1, \ldots, x_n]$ represents a configuration of n parameters as a vector. Estimating the full joint distributions $p_g(\boldsymbol{x})$ and $p_b(\boldsymbol{x})$ over the parameter space is not feasible as it would require significant amount of data. Instead, we simplify by assuming a factorized distribution for both $p_g(\boldsymbol{x})$ and $p_b(\boldsymbol{x})$:

$$p_g(\boldsymbol{x}) = p_{g,x_1}(x_1) p_{g,x_2}(x_2) \ldots p_{g,x_n}(x_n) \tag{7}$$

$$p_b(\boldsymbol{x}) = p_{b,x_1}(x_1) p_{b,x_2}(x_2) \ldots p_{b,x_n}(x_n) \tag{8}$$

Here, each parameter x_i has two corresponding distributions $p_{g,x_i}(x_i)$ and $p_{b,x_i}(x_i)$. Despite this simplification, we found that our method works well in practice.

Note that a particular parameter x_i can be continuous or discrete, and this approach can support both the types as described below.

1) Discrete Parameters: For a discrete parameter, discrete distributions $p_{g,x_i}(x_i)$ and $p_{b,x_i}(x_i)$ are estimated using histograms. $p_{g,x_i}(x_i)$ is estimated by computing a histogram of all the $x_i^* \in \mathcal{H}_t$ such that corresponding $y_i^* < y^{(\tau)}$. Similarly for $p_{b,x_i}(x_i)$ such that corresponding $y_i^* \geq y^{(\tau)}$.

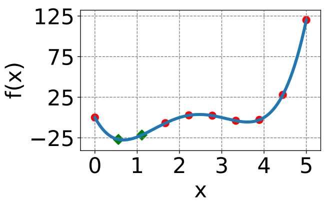
2) Continuous Parameters: For continuous parameters, we use kernel density estimation (KDE) to estimate both the probability density functions. We use gaussian kernels with a fixed bandwidth in our implementation.

C. Putting into Practice

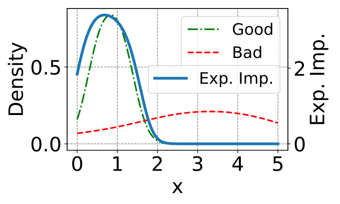
We now put all the steps together and describe the entire algorithm.

1) Select a small set of training samples uniformly at random from the configuration space. Obtain the true objective function value for each by running the experiment. Add all the (sample, value of the objective function) pairs to the list containing the observations ($\mathcal{H}_0$). For our experiments, we selected 20 training samples to add to the initial observations.
2) Build the surrogate model using the list of samples in the observation history. The samples are divided into two groups, good and bad, based on the quantile threshold. The surrogate model then constructs two probability densities ($p_g(\boldsymbol{x})$ for good and $p_b(\boldsymbol{x})$ for bad) for each of the parameter in the configuration domain. We chose 20% as the quantile threshold for our experiments.
3) Select one sample to add to the observation history based on the prediction from the surrogate model (given by equation 5). The selection algorithm selects the best candidate with the highest expected improvement metric.
4) Evaluate the true objective function of the selected candidate by running the experiment. Add the candidate and objective function value pair to the observation history. Update the surrogate model with the new history.
5) Iterate over steps 3-4 until termination condition is met.

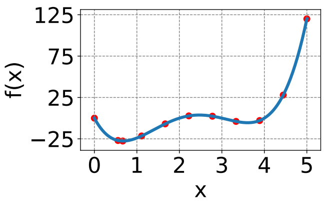
The iterative process adds more and more samples that are expected to be high-performing configurations. This further improves the surrogate model to produce better predictions of the expected improvement. As a result, it progressively



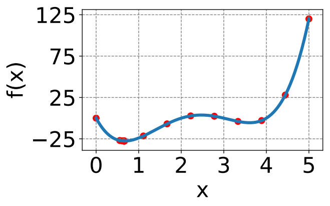
(a) Random Sampling



(b) Surrogate Function



(c) Iter-1



(d) Iter-10

Figure 1: A toy example. 1a shows a simple objective (blue line) along with the initial samples color coded green for good and red for bad. 1b shows the color coded probability density functions ($p_g(\boldsymbol{x})$ and $p_b(\boldsymbol{x})$) constructed using the initial samples along with the Expected Improvement (blue). Figures 1c and 1d show all the samples after one and ten iterations respectively. We can see that the selected samples are concentrating near the minima yielding the optimal parameter value.

selects better and better high-performing configurations. The termination condition can be determined either by the number of objective function evaluations (experimental runs) that can be performed, or based on the quality of the samples obtained as the iteration progresses. If the score of the new samples do not improve as iterations progress, then the iterative process can be terminated.

We provide a toy example to show the main steps involved in the algorithm. Figure 1a (blue line) shows a toy objective function with one parameter. We are interested in finding the parameter value that minimizes the objective. We start with ten training samples selected uniformly at random from this space and add them to the list of observations. A surrogate model is then constructed using the bottom 20^{th} percentile of the observed objective values for setting the threshold for good and bad configurations. Figure 1a plots the initial samples with green for good and red for bad. Figure 1b shows the corresponding probability density functions ($p_g(\boldsymbol{x})$ and $p_b(\boldsymbol{x})$) constructed using the initial samples along with the Expected Improvement (blue). Figures 1c and 1d show all the samples after one and ten iterations respectively. We can see that the selected samples are concentrating near the minima yielding the optimal parameter value.

D. Selection Strategy: Ranking vs. Proposal

When deciding the next set of samples to select based on the surrogate function, there are two options depending on the parameter space. If the parameter space is discrete, finite and small, we can construct an exhaustive set of samples in the space. Expected improvement can then be computed for all of them to pick the best.

However, if the parameter space is continuous then it will not be possible to evaluate an exhaustive set of parameter configurations. In that case, our method can sample candidates using probability density function $p_g(\boldsymbol{x})$ defined for good configurations. This would sample promising candidates while still allowing for exploration due to random sampling.

E. Transfer Learning

When there are limited resources available to collect observations or training data, we can use earlier studies to guide us through the process of modeling the performance. This is common in HPC, where users regularly tune the input parameter values for a large scale application based on their study on a similar but much smaller scale problem. Typically, the smaller scale problem is representative and uses fewer resources but shares run-time characteristics with the large scale target application. This approach is referred to as *transfer learning*, where data from prior studies from a *source domain* is leveraged to guide and accelerate the study on the *target domain*. When the prior studies are relevant, transfer learning can achieve better results with far fewer trials.

We propose to use transfer learning with Bayesian optimization to select high-performing configuration for the *target domain*. Here we will use the data from a *source domain* to improve prediction performance and make rapid progress when modeling the *target domain*. Let D^{Src} be the *source domain* and D^{Trgt} be the *target domain*. The domain consists of common n tunable parameters represented by $x_i, i \in \{1, \ldots, n\}$. The goal of the learning task is to identify high-performing configuration for the *target domain* D^{Trgt}. We use the Bayesian surrogate model for this task. Since the surrogate model maintains probability density functions, we can seamlessly incorporate data from D^{Src} as a prior distribution. Once the prior distribution is applied, we perform the same iterative method as described in section III-C to pick the high-performing samples. The surrogate model probability density function will be a weighted sum of the prior distribution with the distribution from the D^{Trgt}.

$$p_g(x_i) = w * p_g^{Src}(x_i) + p_g^{Trgt}(x_i) \tag{9}$$

$$p_b(x_i) = w * p_b^{Src}(x_i) + p_b^{Trgt}(x_i) \tag{10}$$

The underlying idea is that instead of conducting many resource-heavy runs in the large-scale domain (D^{Trgt}), we use most of the data from small-scale domain (D^{Src}) to select the optimal configuration for D^{Trgt}.

IV. Experimental Setup

This section describes the application datasets used in our case studies, their configurable parameters, and the metrics used for evaluation.

A. Evaluation Datasets

We use several HPC proxy applications as case studies, and use the datasets published in [10] and [11] to evaluate the effectiveness of the proposed approach. More details about the collection and setup can be found in [10], [11]. We use a diverse set of applications consisting of autotuning for compiler flags, application-level parameters, and runtime options (e.g. OpenMP thread count, power cap). Here, we will briefly describe the applications as well as their parameters.

Kripke: Kripke is a proxy application designed for a production code [12], which is a discrete-ordinates S_N deterministic particle transport code. The main parameters for Kripke include options for data layouts, group and energy sets, solver, and number of OpenMP threads. In addition to the application-level parameters, there is a hardware-level configurable parameter for power cap.

HYPRE: HYPRE [13] is a library containing a comprehensive suite of scalable linear solvers for large-scale HPC applications. We use the test benchmark `new_ij` in the HYPRE library for evaluating various application-level tunable parameters. The configurable parameters for HYPRE are solver, smoother, coarsening scheme, and interpolation operator.

We will use Kripke and HYPRE to evaluate both the high-performing configuration selection task as well as transfer learning task. The source dataset for the transfer learning consists of data collected by running the application on smaller node count (16 instead of 64 nodes in the target dataset) with a smaller problem size.

LULESH: LULESH [14] is a proxy application that approximates shock hydrodynamics calculations. LULESH uses several compiler optimization flags as well as OpenMP options to achieve good performance.

OpenAtom: OpenAtom [15] is an application written in Charm++ for studying atomic, molecular, and condensed phase materials systems based on quantum mechanical principles. Charm++ applications rely on over-decomposition of the physical domain to achieve load balancing and overlap of communication with computation. However, over-decomposition of the domain can incur overhead. Hence, it is important to choose the right level of over-decomposition, which is one of tunable parameters we will be studying. In addition, the parameter space includes options for type of density and pair calculation.

B. Evaluation Metrics

A model for selecting high-performing configurations is expected to have several good configurations in the list of samples selected. Moreover, a high-fidelity model would have the best performing configuration or the ones close to the best performing configuration in its list of samples. One option for evaluation metric is to use classification accuracy, which determines the accuracy with which the model is able to correctly classify configurations as optimal or non-optimal. However, that is not applicable in this context because we do not care about accurately selecting non-optimal configurations. Therefore, the metrics for evaluation should focus on how well the model is able to select high-performing configurations.

Configuration Selection Evaluation Metric

The following are the two metrics we use for evaluating methods for configuration selection.

1) **Best Performing Configuration**: This metric records the best performing configuration from the list of selected samples.
2) **Recall**: This metric gives the ratio of number of "good" configurations that were included in the selected set of samples and the actual number of "good" configurations in the entire sample space. Here, a "good" configuration refers to those configurations which are among the best ℓ percentile (the one with the smallest run-time). Let y^ℓ be the objective function value corresponding to the best ℓ percentile configuration. The *Recall* metric is defined as follows:

$$R(\ell) = \frac{|\{x \mid x \in \mathcal{H}, f(x) \leq y^\ell\}|}{|\{x \mid \forall x, f(x) \leq y^\ell\}|} \tag{11}$$

where $|.|$ is the cardinality of a set, $\mathcal{H}$ is the set of configurations selected by the model for which there are observations, and $f(x)$ is the true objective function.

Transfer Learning Evaluation Metric

We use the same *Recall* metric as above for transfer learning. For transfer learning, a "good" configuration refers to that configuration which is within a performance threshold tolerance of $\gamma\%$ from the absolute best performance. We chose this as opposed to top ℓ percentile (used in the configuration selection task) in order to have the same evaluation criteria used in [11]. In [11], good samples are within $\gamma\%$ threshold of the absolute best configuration. Therefore, the *Recall* metric is defined as follows:

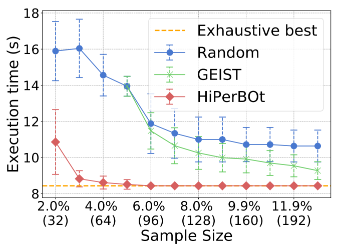
$$R(\gamma) = \frac{|\{x \mid x \in \mathcal{H}, f(x) \leq (1+\gamma) f(x_{best})\}|}{|\{x \mid \forall x, f(x) \leq (1+\gamma) f(x_{best})\}|} \tag{12}$$

where $|.|$ is the cardinality of a set, $\mathcal{H}$ is the set of configurations selected by the model, $f(x)$ is the true objective function, and $f(x_{best})$ is the objective function for absolute best configuration. Higher *Recall* score indicates that a larger number of good configurations are present in the model's selected list. A *Recall* score of 1 implies that the model has selected all the best performing configurations.

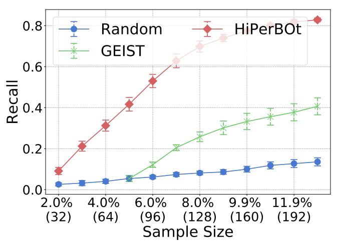
V. Evaluation of Configuration Selection

In this section we evaluate our approach against a few configuration selection methods using the datasets described in section IV-A. For each dataset, we analyze the performance of all the methods for a range of samples by running the model algorithm 50 times and reporting the mean and standard deviation for each evaluation metric. We evaluate the effectiveness of HiPerBOt by comparing against the methods listed below.

1) *GEIST*: This is a semi-supervised learning based adaptive sampling scheme for parameter space exploration. GEIST [10] represents the parameter space using undirected graphs, and applies label propagation method. GEIST bootstraps its search with an initial set of configurations. The nodes of the graph are assigned labels, optimal and non-optimal, based on some initial threshold for the objective function. It uses an iterative approach to propagate the labels to all the nodes in the parameter space graph, and selects a set of optimal configurations based on the CAMLP [16] label propagation algorithm.

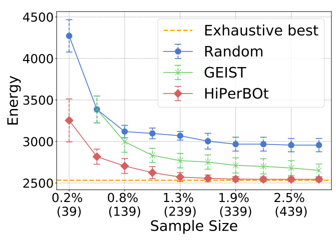


(a) Best Configuration

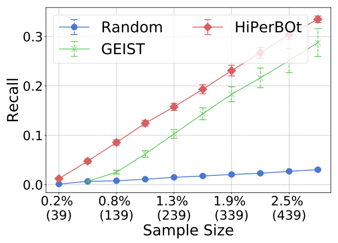


(b) Recall

Figure 2: Kripke execution time. 2a shows that HiPerBOt is able to find the absolute best configuration using just 96 samples, which yields an improvement of 26% in execution time in comparison to GEIST. 2b indicates that HiPerBOt finds more than 2X the number of good configurations in comparison to GEIST.

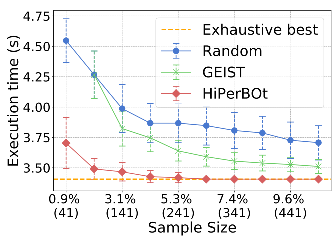


(a) Best Configuration

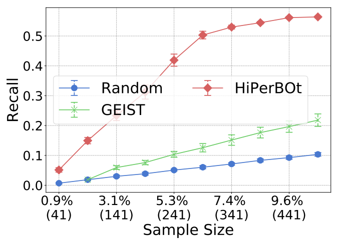


(b) Recall

Figure 3: Kripke energy. 3a shows us that HiPerBOt is able to select the best configuration that achieves the lowest energy by evaluating only 2.2% of the whole sample space. 3b shows that HiPerBOt has a recall score of 0.3, for which, 300 out of 439 samples are good samples.



(a) Best Configuration



(b) Recall

Figure 4: HYPRE. 4a shows that it progressively selects better configurations, and can find the absolute best using 5% of the sample space. 4b indicates that HiPerBOt becomes better at selecting good configurations resulting in a sharp increase in *Recall* score.

2) *Random Selection*: In this strategy the configurations are selected uniformly at random from the parameter space.
3) *Exhaustive Best*: This is the performance of the best configuration, which can be obtained using an exhaustive search of all the samples in the sample space.

GEIST has been compared with other sampling approaches, such as Gaussian Process (GP) [17] and Canonical Correlation Analysis (CCA) [18], and shown to be performing significantly better [10]. Therefore, we do not include results for GP and CCA, and instead just compare with GEIST.

A. *Kripke*

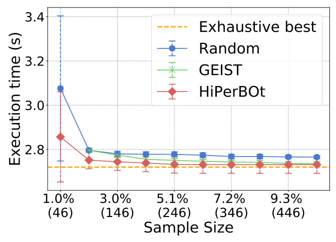
Kripke has several application-level tunable parameters, such as data layout, OpenMP thread count and number of sets to overlap communication with computation. There are a total of 1609 configurations with a large variance in performance metric.

An expert's choice of parameters selection would involve manually testing for each loop ordering with a few group/energy sets, which would result in the execution time of 15.2 seconds [10]. Figure 2a shows that HiPerBOt can find the absolute best configuration with the lowest execution time (8.43 seconds) using just 96 samples, which is less than half the number of samples required by GEIST. Moreover, the runtime of the best performing configuration for HiPerBOt is better than the best configuration for GEIST by 26% for the sample size of 96. Performance of GEIST and random sampling suffers because there are only a few samples in the high-performing bins, which makes it difficult to pick them. Note that as the sample size increases, *Recall* metric also improves. HiPerBOt has more than twice the number of high-performing samples in the sample set in comparison to GEIST.

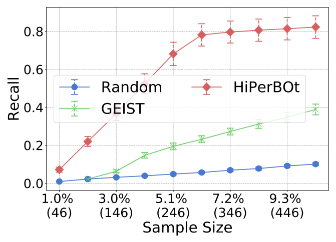
In addition to finding configurations with the least run-time, we can use HiPerBOt to select configurations that minimizes the total energy consumption under power-capping. An expert's choice to optimize for energy would be to use 2^{nd} or 3^{rd} highest power level [10] to achieve energy consumption of 4742 Joules, which is much higher than the best configurations selected by autotuning. Figure 3 shows the performance of HiPerBOt in comparison to GEIST and random sampling. This dataset consists of 17815 configurations, including hardware-level power capping options. HiPerBOt is able to select the best configuration that achieves the lowest energy by evaluating only 2.2% of the whole sample space. This particular dataset has more than 800 good configurations that satisfy the tolerance threshold, which is why Figure 3b has a maximum *Recall* score of 0.3, at which point around 300 good samples are present in the sample set with 489 samples.

B. *HYPRE*

The `new_ij` benchmark of HYPRE has four tunable parameters, namely, solver, smoother, coarsening scheme, and interpolation operator. This generates a total of 4589 configurations. HYPRE has a distribution similar to that of Kripke, where there are only a few samples close to the best performing bins. HiPerBOt is able to pick the best performing configuration as well as have a large number of good configurations in its sample set. We can also see in fig. 4a that it progressively selects better configurations, and can quickly narrow down to the best performing configuration by evaluating just over 5% of the sample space. We observe in fig. 4b that, as iterations progress, the surrogate model becomes better at selecting high-performing configurations, resulting in a better *Recall* score. For a sample size of 241, the run-time for the best configuration for HiPerBOt is 5% better than that of GEIST.

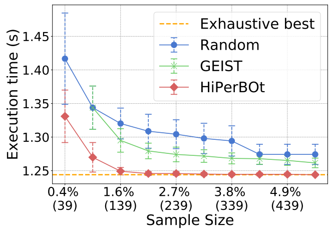


(a) Best Configuration

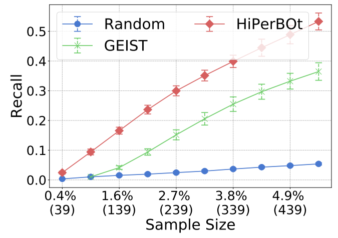


(b) Recall

Figure 5: LULESH. 5b shows a *Recall* score of 0.8 for HiPerBOt, indicating that it finds 80% of the good configurations. This is more than 2X the number of good configurations selected by GEIST.



(a) Best Configuration



(b) Recall

Figure 6: OpenAtom. 6a shows that HiPerBOt is able to select the best performing configuration by evaluating only 3% of the parameter space. 6b indicates that the *Recall* score of HiPerBOt is 30% better than GEIST.

C. LULESH

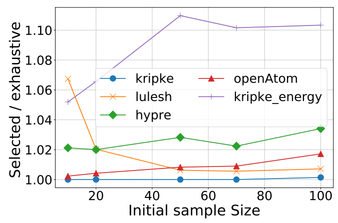
For LULESH, there are eleven compiler flag options available for tuning, resulting in a total of 4800 configurations. Users often resort to using the default flags enabled by the -O3 compiler flag option, which results in an execution time of 6.02 seconds (best is 2.72 seconds). However, -O3 does not necessarily yield best performance. Figure 5b shows that HiPerBOt has more than twice the number of good configurations in its sample set in comparison to GEIST. A *Recall* score of 0.8 for HiPerBOt indicates that 80% of the good configurations have been selected.

D. OpenAtom

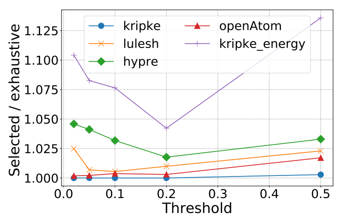
OpenAtom has 8 tunable parameters and a total of 8928 possible configurations. An expert user would apply a symmetric decomposition, which has an execution time of 1.6 seconds (best is 1.24 seconds). We observe in fig. 6a that HiPerBOt is able to select the best performing configuration while exploring only 3% of the parameter space. The *Recall* score of HiPerBOt is 30% better than that of GEIST for the sample size of 489, which means that HiPerBOt has 30% more good configurations than GEIST.

E. HiPerBOt Hyperparameters Sensitivity

In this section we evaluate the sensitivity of the performance with respect to the hyperparameters of HiPerBOt. The two hyperparameters are: 1) number of initial samples used for initializing the model, and 2) quantile threshold for constructing probability densities for $p_g(x)$ (good configurations) and $p_b(x)$



(a) Initial Samples



(b) Percentile Threshold

Figure 7: Performance as a function of the hyper-parameter value. Y-axis shows the ratio of the execution time of the best sample selected by HiPerBOt and execution time of the best sample obtained by exhaustive evaluation. Higher value of the ratio indicates poor selection while a value close to one indicates near optimal selection. 7a shows the performance of HiPerBOt for different initial number of samples. Initial sample size of 20 gives the best performance for most of the applications. 7b shows the performance of HiPerBOt for different thresholds,where threshold of 0.20 percentile shows the best performance.

(bad configurations). To analyze the sensitivity we report the performance in terms of the ratio of the execution time of the best sample selected by HiPerBOt and execution time of the best sample obtained by exhaustive evaluation as a function of the hyperparameter value. Higher value of the ratio indicates poor selection while a value close to one indicates near optimal selection. Figure 7a visualizes the evaluation results for the number of initial samples. The initial sample size is varied from 10 to 100, and the total number of samples is fixed to 150. Figure 7b visualizes the results for the quantile threshold. The quantile threshold is varied from 0.01 to 0.5. Note that the performance for initial sample size of 100 is slightly worse than others because only 50 more samples are selected that are based on the model. The evaluation shows that the performance is less sensitive to the initial sample size for most of the applications, while there is a sweet spot for the percentile threshold.

VI. Parameter Importance Analysis

A particular choice of the value for a parameter can significantly impact the application performance metrics. However, the performance metrics are not equally sensitive to all the parameters. Some parameters tend to have a more pronounced impact on the application performance in comparison to others. Here, we propose to use the surrogate model to quantify the effect of different parameters on the performance.

The surrogate model maintains two distributions for each parameter x_i: 1) $p_{g,x_i}(x_i)$ is the distribution over x_i corresponding to good configurations, and 2) $p_{b,x_i}(x_i)$ for bad configurations. Our key observation is that for the parameters with significant impact on the application performance, the set of values corresponding to good performance will be different from those for bad performance. Therefore, the distributions $p_{g,x_i}(x_i)$ and $p_{b,x_i}(x_i)$ will be significantly different from each other. We propose to use the difference between the two distributions as a measure of the relative importance of a parameter.

There are a variety of choices for computing the difference

TABLE I: Relative ranking of parameters.

Application	10% samples	All samples
HYPRE	Solver(0.12),Ranks(0.08),OMP(0.02), Smoother(0.01),MU(0.00),PMX(0.00)	Ranks(0.49),OMP(0.32),Solver(0.26), Smoother(0.01),MU(0.00),PMX(0.00)
OpenAtom	rhory(0.08),sgrain(0.07),rhorx(0.04), gratio(0.04),rhoratio(0.03),rhohx(0.01), rhohy(0.01),ortho(0.00)	sgrain(0.26),rhory(0.08),gratio(0.08), rhohx(0.04),rhohy(0.03),rhorx(0.02), rhoratio(0.01),ortho(0.00)
Kripke exec	Ranks(0.11),Nesting(0.08),Gset(0.08), Dset(0.06),OMP(0.02)	Ranks(0.21),OMP(0.12),Dset(0.08), Gset(0.08),Nesting(0.05)
Kripke energy	Nesting(0.17),PKG LIMIT(0.07),Gset(0.06), Ranks(0.02),Dset(0.00),OMP(0.00)	Nesting(0.16),Ranks(0.16),OMP(0.13), PKG LIMIT(0.12),Gset(0.09),Dset(0.03)
LULESH	level(0.10),malloc(0.09),force(0.06), builtin(0.05),unroll(0.05), noipo(0.00), strategy(0.00),functions(0.00)	builtin(0.21),malloc(0.17),unroll(0.13), level(0.04),force(0.03), noipo(0.01),strategy(0.00),functions(0.00)

between two distributions. In this work, we propose to use the Jensen-–Shannon (JS) divergence $D_{JS}(p_{g,x_i}(x_i), p_{b,x_i}(x_i))$ for its symmetry in arguments. For two probability distributions P and Q defined in the same probability space X, the JS divergence is defined using $M = (P + Q)/2$ as

$$D_{JS}(P, Q) = \frac{1}{2} D_{KL}(P, M) + \frac{1}{2} D_{KL}(Q, M) \tag{13}$$

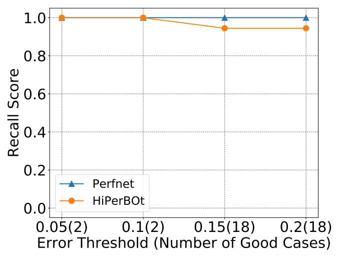
$$D_{KL}(P, M) = \sum_{x \in X} P(x) \log \frac{P(x)}{M(x)} \tag{14}$$

Here $D_{KL}(P, M)$ is the Kullback–Leibler (KL) divergence from M to P. $D_{KL}(Q, M)$ is defined similarly. Note that $D_{JS}(P, Q) \geq 0$, with equality for identical distributions.

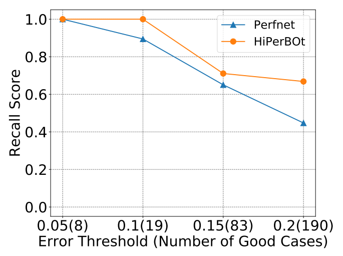
Table I shows the relative importance of the parameters for various applications studied in the paper. We show all the parameters for each application along with the JS divergence when 10% samples are used to construct the surrogate model as well as when all the samples are used (actual ranking). For HYPRE, the combination of number of MPI ranks and OpenMP threads per node affects resource utilization and application time. Therefore, we expect the application performance to be sensitive to those parameters. In addition, the type of solver also affects the application performance. For LULESH, except for the compiler flags strategy, noipo, and functions, all the compiler parameters are critical in determining good performing configurations. For Kripke, we notice that the ranking varies slightly between the surrogate model as well as the actual ranking using all the data. We hypothesize that this is due to the fact that all the parameters have significant impact on the performance. As evident, our method is able to identify important parameters with only a fraction of the total number of samples in most of the applications.

VII. Evaluation of Transfer Learning

We evaluate HiPerBOt against PerfNet [11], a recently published transfer learning approach used in HPC domain. PerfNet is a deep learning based transfer learning approach that combines observations at smaller scale with limited observations collected at larger scale. For transfer learning, we use all the data from $\mathcal{D}^{Src}$ to act as the prior distribution for our surrogate model. Note that we use the same number of samples as used for the evaluation of PerfNet in order to reuse the results published in [11]. The number of samples selected is



(a) Kripke



(b) HYPRE

Figure 8: Kripke: 8a shows that for thresholds 5% and 10%, both PerfNet and HiPerBOt are able to find all the good configurations. For 15% and 20% HiPerBOt finds 17 out of 18 good configurations.HYPRE: 8b shows that HiPerBOt achieves a better *Recall* score in comparison to PerfNet. For tolerance threshold of 5% HiPerBOt finds all the good configurations.

1% of the total samples in $\mathcal{D}^{Trgt}$ plus 100 more. The evaluation is conducted for different performance tolerances, including $5\%, 10\%, 15\%$, and 20%. We use Kripke and HYPRE for our study.

A. *Kripke*

Kripke has several tunable parameters for selecting the data layouts to improve parallelism and in turn performance. There are 17815 configurations in the $\mathcal{D}^{Src}$ dataset, and 17385 configurations in the $\mathcal{D}^{Trgt}$. All the data from $\mathcal{D}^{Src}$ are used to construct the prior probability densities. Total of 273 samples are selected from $\mathcal{D}^{Trgt}$, and HiPerBOt iteratively selects high-performing samples to add to the list of observations. Figure 8a shows the *Recall* score for PerfNet and HiPerBOt. For small performance thresholds of 5% and 10%, both PerfNet and HiPerBOt are able to select all the good configurations resulting in a *Recall* score of 1.0. For performance thresholds of 15% and 20%, *Recall* of HiPerBOt is slightly lower than that of PerfNet; however, it selects 17 out of the 18 good configurations.

B. *HYPRE*

The `new_ij` benchmark for HYPRE has parameters to select the type of solver, smoother etc. The total number of configurations in the $\mathcal{D}^{Src}$ is 57313 and that in $\mathcal{D}^{Trgt}$ is 50395. Again, we use all the data from the $\mathcal{D}^{Src}$ domain as prior for the surrogate model. Here, the models pick 603 samples from $\mathcal{D}^{Trgt}$ to evaluate. Figure 8b shows the comparison of HiPerBOt with PerfNet. We observe that PerfNet has a *Recall* score of 1.0 for performance tolerance threshold of 5%, but the score starts to decrease as the tolerance increases. The *Recall* score drops with increase in tolerance threshold because there are more configurations that qualify as good configurations, but the total samples selected remains 603. HiPerBOt is able to maintain the *Recall* score at 1.0 identifying all the 19 configurations for tolerance threshold of 10%, and achieves a higher *Recall* score than PerfNet.

In summary, irrespective of the distribution of samples, HiPerBOt is very good at identifying high-performing configurations, and selecting the absolute best configuration with very limited samples for all the benchmarks. The runtime for

HiPerBOt is significantly less than the application time for a single configuration. For example, HiPerBOt for LULESH took around 600 ms to select the best configuration whereas evaluating all configurations took more than 19 hours (as reported in [10]) and the best application time was 2.7 seconds. Moreover, HiPerBOt consistently outperforms GEIST for configuration selection, and achieves better results in comparison to PerfNet for transfer learning.

VIII. Related Work

The traditional method for tuning scientific libraries and applications uses analytical methods to predict performance. Development of these analytical models requires expert knowledge of the underlying architecture as well as application characteristics. Moreover, these methods tend to have poor accuracy when evaluating complicated systems [19]. Alternatives to analytical methods are empirical performance models, which rely on experimental evaluations on target machines to build performance prediction models [1], [20]–[24]. Bergstra et al. [2] used boosted regression trees for creating a performance model. Balaprakash et al. [3] presented an automatic multi-objective modeling approach for predicting application performance and power usage based on hardware performance counters. Beckingsale et al. [1] used decision tree based classifier model to select runtime execution policies. While these supervised learning based methods can be used for parameter space tuning of HPC applications, they require a large amount of training data which can incur a high cost. These training instances are randomly selected to evaluate irrespective of how useful each instance is in achieving the objective. This leads to a wastage of resources by exploring regions of the parameter space that do not contribute to the goal of the task. Note that the task of selecting high-performing configurations does not require models to perform well for the entire configuration space, enabling our method to perform well by using fewer but informative instances.

Active learning based methods provide a low-cost predictive model with less training overhead by selecting only useful training samples. Ogilvie et al. [25] presented an online predictive algorithm to select training samples. Chen et al. [26] presented a similar online approach for non-HPC applications. Balaprakash et al. [22] proposed an iterative parallel algorithm that builds surrogate performance models using dynamic tree model. Adaptive sampling-based works [17], [18] are very relevant to our approach. Ganapathi et al. [18] proposed a Kernel Canonical Correlation Analysis (KCCA) based approach to derive the relationship of parameters with performance and energy. Duplyakin et al. [17] presented a Gaussian Process Regression based method to minimize search space.Thiagarajan et al. [10] proposed GEIST, an adaptive sampling technique that uses label propagation algorithm to identify well performing parameter configurations. In this paper, we present a detailed comparison of our approach with GEIST and show that our method outperforms their approach. We did not include comparison results with GP [17] and CCA [18] because GEIST was shown to perform significantly better in comparison to them.

Bergstra et al. [5] proposed a Bayesian optimization method, Tree Parzen Estimator (TPE), for optimal parameter selection for training neural networks. Several Bayesian optimization tools, such as, Hiperopt [27], SigOpt [28], Spearmint [29], are geared towards machine learning workloads, where the candidates are sampled from a distribution and are especially suitable when the parameter space is continuous. This work is similar to several of these tools, but differs in that we apply HiPerBOt to performance tuning of HPC applications. Configuration parameters for HPC applications are mostly discrete and finite. We proposed *Ranking* strategy, where the next set of samples is selected by computing the expected improvement for all the samples and picking the best. This also eliminates the scenario where duplicate samples are selected. Moreover, we extend the same Bayesian optimization technique for transfer learning.

Transfer Learning Methods: Transfer learning methods aim to improve the performance or data requirements for a target application by transferring the knowledge from a similar or related application. Roy et al. [30], [31] presented techniques for auto-tuning by porting the performance models created on one architecture and transferring them to other architectures. Muralidharan et al. [32] and Ding et al. [33] performed code-variant tuning using learning based methods. Price et al. [34] optimized the configuration space search by optimizing sample search on progressively larger sets of target platforms. Falch et al. [35] fine-tuned the parallelizing runtime system for target applications on GPU platforms. Grebhahn et al. [36] and Marathe et al. [11] use transfer learning for transferring domain knowledge learned on low-cost configurations to select high-performing configurations for a target application. In contrast, our approach can use just the samples collected for the target problem, or it can use the data from source domain to speedup the learning process in the target domain. An advantage of HiPerBOt is that it can work seamlessly in both the cases.

IX. Conclusion

In this work, we have presented HiPerBOt, an active learning framework based on Bayesian optimization, to select high-performing configurations using limited samples. We showed that HiPerBOt performs significantly better than alternative methods by identifying best-performing configuration with fewer samples. For example, it uses 50% fewer samples for Kripke in comparison to the best alternative method. HiPerBOt also finds more than $2\times$ the number of good configurations for LULESH in comparison to the best alternative. We applied HiPerBOt to transfer learning, where executions at a smaller scale were used to speedup the parameter space tuning process at larger scale. HiPerBOt outperformed the alternative method for Kripke. Most importantly, HiPerBOt provides an efficient way for users to apply autotuning to select high-performing parameter configuration for their application with significantly less resource overhead.

Acknowledgment

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-772119). This work was supported in part by funding provided by the University of Maryland College Park Foundation.

References

[1] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017.

[2] J. Bergstra, N. Pinto, and D. Cox, "Machine learning for predictive auto-tuning with boosted regression trees," in *Proceedings of Innovative Parallel Computing*, May 2012, pp. 1–9.

[3] P. Balaprakash, A. Tiwari, S. M. Wild, L. Carrington, and P. D. Hovland, "Automomml: Automatic multi-objective modeling with machine learning," in *International Conference on High Performance Computing*. Springer, 2016, pp. 219–239.

[4] J. Mockus, *Bayesian approach to global optimization: theory and applications*. Springer Science & Business Media, 2012, vol. 37.

[5] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011, pp. 2546–2554.

[6] D. R. Jones, "A taxonomy of global optimization methods based on response surfaces," *Journal of global optimization*, vol. 21, no. 4, 2001.

[7] F. Hutter, "Automated configuration of algorithms for solving hard computational problems," Ph.D. dissertation, University of British Columbia, 2009.

[8] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International Conference on Learning and Intelligent Optimization*. Springer, 2011.

[9] J. Villemonteix, E. Vazquez, and E. Walter, "An informational approach to the global optimization of expensive-to-evaluate functions," *Journal of Global Optimization*, vol. 44, no. 4, p. 509, 2009.

[10] J. J. Thiagarajan, N. Jain, R. Anirudh, A. Gimenez, R. Sridhar, A. Marathe, T. Wang, M. Emani, A. Bhatele, and T. Gamblin, "Bootstrapping parameter space exploration for fast tuning," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: ACM, 2018, pp. 385–395. [Online]. Available: http://doi.acm.org/10.1145/3205289.3205321

[11] A. Marathe, R. Anirudh, N. Jain, A. Bhatele, J. Thiagarajan, B. Kailkhura, J.-S. Yeom, B. Rountree, and T. Gamblin, "Performance modeling under resource constraints using deep transfer learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 31:1–31:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126969

[12] A. Kunen, T. Bailey, and P. Brown, "KRIPKE-a massively parallel transport mini-app," *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2015.

[13] R. D. Falgout and U. M. Yang, "HYPRE: A Library of High Performance Preconditioners," in *Computational Science–ICCS 2002*. Springer, April 2002, pp. 632–641.

[14] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-641973, August 2013.

[15] N. Jain, E. Bohm, E. Mikida, S. Mandal, M. Kim, P. Jindal, Q. Li, S. Ismail-Beigi, G. J. Martyna, and L. V. Kale, "Openatom: Scalable ab-initio molecular dynamics with diverse capabilities," in *International Conference on High Performance Computing*. Springer, 2016.

[16] Y. Yamaguchi, C. Faloutsos, and H. Kitagawa, "Camlp: Confidence-aware modulated label propagation," in *Proceedings of the 2016 SIAM International Conference on Data Mining*. SIAM, 2016, pp. 513–521.

[17] D. Duplyakin, J. Brown, and R. Ricci, "Active learning in performance analysis," in *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*. IEEE, 2016, pp. 182–191.

[18] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A case for machine learning to optimize multicore performance," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, 2009.

[19] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 879–892.

[20] I.-H. Chung and J. K. Hollingsworth, "A case study using automatic performance tuning for large-scale scientific programs," in *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*. IEEE, 2006, pp. 45–56.

[21] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Minimizing the cost of iterative compilation with active learning," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 245–256.

[22] P. Balaprakash, R. B. Gramacy, and S. M. Wild, "Active-learning-based surrogate models for empirical performance tuning," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference*, 2013.

[23] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[24] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014.

[25] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Fast automatic heuristic construction using active learning," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2014.

[26] J. K. Chen, R.-B. Chen, A. Fujii, R. Suda, and W. Wang, "Surrogate-assisted tuning for computer experiments with qualitative and quantitative parameters," 2017.

[27] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International Conference on Machine Learning*, 2013, pp. 115–123.

[28] I. Dewancker, M. McCourt, and S. Clark, "Bayesian optimization primer," 2015.

[29] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[30] A. Roy, P. Balaprakash, P. D. Hovland, and S. M. Wild, "Exploiting performance portability in search algorithms for autotuning," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016.

[31] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: An exploratory analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[32] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A Framework for Adaptive Code Variant Tuning," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, May 2014, pp. 501–512.

[33] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, Jun. 2015.

[34] J. Price and S. McIntosh-Smith, "Improving auto-tuning convergence times with dynamically generated predictive performance models," in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*. IEEE, 2015, pp. 211–218.

[35] T. L. Falch and A. C. Elster, "Machine learning-based auto-tuning for enhanced performance portability of opencl applications," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 8, 2017.

[36] A. Grebhahn, N. Siegmund, H. Köstler, and S. Apel, "Performance prediction of multigrid-solver configurations," in *Software for Exascale Computing*. Springer, 2016, pp. 69–88.