Integrating Bayesian Optimization and Machine Learning for the Optimal Configuration of Cloud Systems

Bruno Guindani, *Graduate Student Member, IEEE*, Danilo Ardagna, *Senior Member, IEEE*, Alessandra Guglielmi, Roberto Rocco, *Graduate Student Member, IEEE*, and Gianluca Palermo, *Senior Member, IEEE*

***Abstract*—Bayesian Optimization (BO) is an efficient method for finding optimal cloud configurations for several types of applications. On the other hand, Machine Learning (ML) can provide helpful knowledge about the application at hand thanks to its predicting capabilities. This work proposes a general approach based on BO, which integrates elements from ML techniques in multiple ways, to find an optimal configuration of recurring jobs running in public and private cloud environments, possibly subject to black-box constraints, e.g., application execution time or accuracy. We test our approach by considering several use cases, including edge computing, scientific computing, and Big Data applications. Results show that our solution outperforms other state-of-the-art black-box techniques, including classical autotuning and BO- and ML-based algorithms, reducing the number of unfeasible executions and corresponding costs up to 2–4 times.**

***Index Terms*—Acquisition function, Bayesian optimization, black-box optimization, machine learning.**

I. Introduction

TODAY, Information and Communication Technology (ICT) systems often exploit increasingly complex applications. These applications can execute in a distributed fashion on computer networks or cloud systems and are often computing intensive. This complexity allows little insight into their inner workings, especially for system administrators who did not engineer them. Examples of these systems include Big Data analytic tools running on the cloud, scientific computing programs for simulations requiring massive computational power, electronic or mechanical devices operated by Artificial Intelligence, and computing continuum systems for efficient distributed computation. These systems require the configuration of software settings or the available hardware resources (CPU, memory, disk, network, etc.). However, a poor choice for such settings can lead to application under-performance or additional costs for the end users. The impact of incorrect configuration is potentially paramount [1], [2], [3], and proper optimization prevents such negative effects. For instance, running a big-data application in the cloud without the correct configuration multiplies the execution cost by a factor of three on average, up to ten in the worst case [3]. This is especially true for recurring jobs, i.e., applications that must execute multiple times, possibly with regular frequency. In this case, the additional cost of suboptimal configurations adds up over time.

Furthermore, it is often crucial that the execution of an application complies with given requirements, either coming from the service provider as part of its business model (e.g., providing several performance targets/Service Level Objectives) or by the customers' needs. However, choosing the best configuration that satisfies a-priori fixed constraints is challenging due to the diverse behavior and resource requirements of cloud systems.

These limitations make it difficult to effectively employ white-box methods such as Petri nets [4] and queueing networks [5] to study software performance and compliance with constraints. The low-level information and metrics required to parametrize white-box models are often hard to measure, if not straight-up impossible to access, for complex programs running in partially accessible environments such as cloud data centers and High-Performance Computing (HPC) infrastructures. Even if such information were available, relations among input workloads, program configurations, internal workflows, and specifics of ICT systems would follow complex patterns, which would be difficult to express with analytical models [3]. Regardless, it would be impossible to formulate one single model to cover all use cases. Therefore, one must create multiple individual, domain-specific models, each requiring thorough domain expertise, extensive effort, and significant profiling costs [6]. Such an approach is hardly scalable to large computing environments, applications, and domains.

For these reasons, general approaches that do not require any knowledge of the internal details of the system are becoming popular. The literature generally refers to such approaches as

Manuscript received 13 February 2023; revised 2 January 2024; accepted 28 January 2024. Date of publication 1 February 2024; date of current version 8 March 2024. The work of Alessandra Guglielmi was supported by MUR, Grant Dipartimento di Eccellenza 2023-2027. This work was supported by the European Commission through the Horizon 2020 under Grant 956137 LIGATE: LIgand Generator and portable drug discovery platform AT Exascale, as part of the European High-Performance Computing (EuroHPC) Joint Undertaking Program. Recommended for acceptance by D. Wu. *(Corresponding author: Bruno Guindani.)*

Bruno Guindani, Danilo Ardagna, Roberto Rocco, and Gianluca Palermo are with the Department of Electronics, Information, and Bioengineering, Politecnico di Milano, 20133 Milano, Italy (e-mail: bruno.guindani@polimi.it; danilo.ardagna@polimi.it; roberto.rocco@polimi.it; gianluca.palermo@polimi.it).

Alessandra Guglielmi is with the Department of Mathematics, Politecnico di Milano, 20133 Milano, Italy (e-mail: alessandra.guglielmi@polimi.it).

Digital Object Identifier 10.1109/TCC.2024.3361070

black-box techniques and the corresponding optimized quantities as black-box functions.

Bayesian Optimization (BO) has recently gained notoriety as a powerful tool to solve global optimization problems that involve expensive, black-box functions [7], [8], [9]. BO is a sequential design strategy that only needs a few steps to get sufficiently close to the true optimum and requires no derivative information on the analyzed function. Most commonly, it starts by choosing and evaluating a handful of starting points, and then using them to fit a Gaussian Process model. The fitted model estimates the function value at each point and the uncertainty around the prediction. The BO method then iteratively chooses new points at which to evaluate the function in such a way as to balance exploration (large uncertainty) and exploitation (best expected value) [7]. The traditional BO solution perfectly suits black-box constrained optimization problems such as cloud configuration. However, most existing approaches fail to adequately take constraints into account, if at all, resulting in many unfeasible configurations chosen.

Machine Learning (ML) models are another popular black-box tool in ICT domains. Their remarkable predicting capabilities can assist in accurately predicting resource usage, execution times, etc., without a detailed knowledge of the system under study. Indeed, previous work [10], [11], [12], [13] showed that ML models usually can predict these target quantities with minimal validation errors. In cloud systems, these predictive abilities can enhance early detection (and prevention) of configurations that breach constraints, potentially leading to substantial savings in time, finances, and other valuable resources.

This work proposes MALIBOO (MAchine Learning In Bayesian OptimizatiOn), a framework integrating BO algorithms with ML techniques to minimize the execution costs of recurring resource-constrained computing jobs. In particular, we present an extension of an earlier work [14], which focused exclusively on a class of one-dimensional optimization problems with a linear relationship between the constraint and the target. New contributions include: 1) generalizing the theoretical framework to fit almost any constrained optimization problem, 2) considering additional scenarios for our experimental campaign, including some covered by this new generalization, and 3) broadening the comparison against the state of the art.

Our contribution is the introduction of a BO-based technique for cloud optimization that supports both black-box constraints and objectives. As shown in the literature review, existing works on BO, cloud optimization, and constrained optimization have restrictions that make them inapplicable to the scenarios we consider. Previous attempts at integrating BO and ML also have restrictions or exploit ML in a limited way. Our algorithm applies to several optimization scenarios, including but not limited to cloud optimization. However, it is especially suitable for the cloud case, in which evaluation of a single configuration is expensive and thus should be avoided if not strictly necessary.

We validate our approach in many scenarios of interest, on both public and private cloud systems, to prove the generality of our solution. The scenarios analyzed include Big Data analytics, edge computing, and scientific computing applications. These applications act as representatives of systems commonly employed in the industry. In the experimental campaign, we compare our algorithm against state-of-the-art black-box techniques, some based on BO and ML, some based on traditional autotuning methods. We observe that the proposed BO algorithm variants reduce both the unfeasible executions and the percentage of unfeasible costs compared to those techniques: up to 2–3 times compared to a classical BO approach and up to 3–4 times to another prominent algorithm integrating BO and ML.

This paper is structured as follows. Section II surveys the state-of-the-art techniques for cloud systems performance analysis and optimal configuration. In Section III, we formalize the problem of optimal application configuration at hand. Section IV presents an overview of BO and explains its key components. Section V details our contributions involving the integration of ML into BO. We explain the settings of our experimental validation in Section VI, present results in Section VII, and discuss them in Section VIII. Conclusive remarks follow in Section IX.

II. Related Work

This section reviews the state of the art for the automatic configuration of general ICT and cloud-based systems.

Several autotuning software solutions allow the exploration of search spaces with varying complexity, although they often cannot consider constraints. OpenTuner [15] is a popular Python-based framework for building domain-specific multi-objective program autotuners. It exploits an ensemble of search techniques running simultaneously and sharing results, while a root meta-technique (e.g., multi-armed AUC bandit) handles the allocation of tests to the various approaches. Optuna [16] is another commonly used software, implementing several kinds of search algorithms (e.g., Random Search, NSGA-II, Quasi-Monte Carlo) and allowing dynamic parameter space construction. However, it targets unconstrained optimization and does not inherently support the presence of constrained resources.

We now examine BO literature for optimal configuration selection and autotuning in Section II-A and ML literature for predicting configuration behaviors in Section II-B. Then, in Section II-C, we also review previous attempts at integrating BO with ML techniques.

A. Bayesian Optimization

In the literature on BO and its application to optimal ICT system configuration, the *CherryPick* framework [3] has been proposed as a solution for cloud optimization. In particular, it utilizes pure constrained BO to find optimal cloud configurations for Apache Spark Big Data applications. Similarly, authors of [2] leverage low-level performance metrics to enhance BO efficiency in finding the best type of Virtual Machine (VM) on the cloud. However, the proposed approach is only applicable whenever such essential system metrics are available. Other works use some BO variants to optimize cloud configuration for automated laboratory instruments [17] and for microservice-based applications [18] – but both approaches do not account for constraints.

Another noteworthy example of cloud systems optimization via BO is Google Vizier [19]. In this work, Google researchers present the internal framework used in their data centers to conduct black-box optimization of physical and software systems running in a production environment. Their default optimization algorithm exploits BO techniques based on batched Gaussian Process (GP) bandits. A work using similar methods is [20], in which authors tackle the problem of selecting the optimal configuration for a database management system. They present an autotuning algorithm built on top of BO, exploiting contextual GP bandits. Yet again, these techniques do not effectively consider constraints. The former requires specifying a list of feasible configurations before the optimization process, while the latter uses a soft constraint approach by incorporating the scarce resource into the objective function.

BO techniques also allow tuning the many hyperparameters of large prediction models, such as Deep Neural Networks (DNN), characterized by a black-box nature and hefty computational costs. We discuss an example of such application in Section II-C.

Hyperopt [21] is a popular example among BO-based autotuning solutions for hyperparameter tuning in ML libraries such as Python Scikit-learn or Spark MLlib. It uses a custom BO algorithm based on Trees of Parzen Estimators, replacing parametric distributions for software configurations with non-parametric densities. Other examples include Kernel Tuner [22], which is specific to GPU applications, and SMAC3 [23], which features both traditional GP-based algorithms and other approaches such as Random Forest-based BO. These libraries do not support constraints either.

Besides cloud optimization [24], BO has been successfully applied to several black-box optimization problems [7] in an unconstrained setting, from energy minimization in molecule simulation [25] to laser-plasma particle accelerators [26]. The topic of constrained BO has also received attention on its own, including works on unknown or black-box constraints. For instance, authors of [8] suggest the Expected Barrier acquisition function, which combines convergence guarantees of numerical local search methods with the global breadth of search provided by BO. The ADMBBO algorithm in [27] turns a constrained optimization problem into an unconstrained one via a Lagrange-esque penalization for constraint violation, then divides it into subproblems, each solved via unconstrained BO. Finally, [28] introduces a probabilistic approach to compliance with constraint functions. The core idea is to introduce a probability-weighted expected improvement acquisition function, in which the user can provide a confidence threshold for the probability of constraint compliance. However, all these methods assume independence of their surrogate models on the target and the constrained function, which is unrealistic in many scenarios, including the ones in our experimental analysis (see, e.g., Section VI-A1).

As mentioned, most other works, especially those on cloud optimization, do not consider constraints. CherryPick [3] does allow them, but the assumptions on its underlying model restrict it to a limited class of constrained optimization problems, as discussed later.

B. *Machine Learning*

ML is a common tool for predicting the performance of ICT systems, such as Big Data applications, training of deep learning models, and Function-as-a-Service (FaaS) systems. Overall, results found in the literature are promising in showing the usefulness of ML in the context of cloud configuration. For instance, [11] examines the performance of several ML models in carrying out predictions of Apache Spark jobs execution times with different types of workloads. Their results outperform the models used by Spark creators. The Hemingway framework [29] specializes in the modeling and identification of optimal cluster configuration for Spark MLlib-based applications. Authors of [12] use several ML models alongside anomaly detection techniques to properly configure a cloud-based Internet of Things (IoT) device manager, while respecting Quality of Service (QoS) constraints. Another recent work [10] explores performance prediction of GPU-deployed neural network training times starting from cloud specifications using ML techniques and feature selection methods. Similarly, [13] compares different popular ML techniques applied to a workload prediction analysis on HTTP servers, showing that they all achieve good predicting capabilities. Finally, the Schedulix framework proposed in [30] uses linear regression to estimate execution latencies of serverless applications in a public cloud FaaS setting.

In general, one must pair the performance prediction capabilities of ML under a given configuration with an effective way to explore and choose them, which is a challenging problem in and of itself.

C. *Integration of Bayesian Optimization and Machine Learning*

Recent BO works involving ML include [31], [32], and [33]. In particular, [31] presents the Paprika scheduler, which co-optimizes hardware and software configurations for Spark workloads. This process uses a BO-based algorithm integrated with ML elements, namely feature selection via Lasso regression. Their model is one of the closest to our research goal, but it requires offline training before deployment, which may only sometimes be possible in practice. Furthermore, their exploitation of ML is arguably limited in scope since they only use it to choose among the existing features without any regression strategy involved. In addition, it does not take constraints into account.

The SVM-CBO algorithm [32] exploits ML to find optimal configurations in a constrained setting, for instance, when deploying DNNs to tiny microcontroller-based systems. Their algorithm consists of two phases: the first phase approximates the feasible domain via a Support Vector Machine (SVM) classification model, while the second phase applies pure BO to the approximated domain found earlier. This approach is arguably inefficient in exploiting the iteration budget since it handles the domain approximation and optimization phases separately.

Finally, [33] introduces the Lynceus framework to jointly optimize software and hardware configuration of data analysis and ML jobs on cloud platforms when subject to time constraints. It simulates exploration paths to assess the long-term impact of the choice of a configuration. It uses a BO method based on an unconventional Decision Tree bagging ensemble as its prior distribution to model the probability of satisfying the constraints.

However, literature shows that Gaussian Process-based BO (like the one present in this work) usually performs better than other BO variations for metrics such as execution time and cost [24].

III. Problem Formulation

We aim to find the optimal configuration for cloud systems under a black-box cost metric and subject to black-box performance and resource constraints, such as execution time or quality thresholds.

We consider the mathematical formulation for our constrained, noisy global optimization problem as follows. Let $x \in \mathcal{A}$ denote the d-dimensional vector representing a configuration for the system at hand, with $\mathcal{A} \subset \mathbb{R}^d$ being the domain of all possible configurations. The numerical vector x includes the values of a (possibly large) number of software and hardware configuration parameters, e.g., the buffer size, the number of total cores (potentially available on multiple homogeneous VMs or physical servers) used for the job, and other application-specific parameters such as algorithm iterations or simulated molecules number. The black-box *objective function* $f(x) : \mathcal{A} \to \mathbb{R}$ to minimize typically measures the performance or quality of configuration x (e.g., a cross-validation error score), the running time or cost of an application, and so forth. Therefore, in all practical applications of interest, $f(\cdot)$ is always a lower-bounded function. We also assume a scalar *constraint function* $g(x) : \mathcal{A} \to \mathbb{R}$, which is possibly independent of the objective function. The constraint $g(\cdot)$ is also a black-box function whose expression is unknown. Therefore, the feasible part of the domain, i.e., the one containing points that satisfy the constraint, is also unknown. We represent such constraint on x with a potentially unbounded closed interval, whose extrema $G_{\min}$ and $G_{\max}$ can be infinite (i.e., $G_{\min}, G_{\max} \in [-\infty, +\infty]$). This assumption is different from other works such as CherryPick [3], which requires strict assumptions on its optimization problem (see also Section VI-A1). The generality of the constraint is also one of the main novel contributions to our previous work [14]. We then have:

$$\begin{aligned} &\min_{x\in\mathcal{A}} f(x) + \varepsilon \\ &\text{s.t.} \quad g(x) \in [G_{\min}, G_{\max}], \end{aligned} \tag{1}$$

where ε is a noise term. Including noise in the model proves necessary in a general formulation because of its usefulness in some specific scenarios. For instance, in a cloud setting, it accounts for the intrinsic variability of an application execution time, even when the application runs multiple times with the same configuration. This variability typically comes from external causes, such as concurrent access to the underlying physical resources or network congestion.

The problem formulation in equation (1) is general enough to model a wide variety of cloud optimization scenarios, ranging from neural network hyperparameter tuning to efficient energy consumption. While we include a single constraint in equation (1), our approach can easily extend to an arbitrary number of constraints $[g_1, \ldots, g_m]$.

IV. Bayesian Optimization Overview

BO is a particularly efficient method within the setting described in Section III because it approximates the minimum of a given black-box objective function $f(\cdot)$ on its domain $\mathcal{A}$ by using as few iterations as possible. Strong assumptions on $f(\cdot)$ or $\mathcal{A}$ are not required; in particular, BO algorithms do not need derivative information. For these reasons, several works use BO methods to optimize expensive black-box objective functions, that is, functions for which little information is available and whose evaluation has significant time, resource, or monetary costs [7].

We now give a brief overview of the main tools used by BO. In particular, we explain the peculiarities of *Gaussian processes* (Section IV-A) and *acquisition functions* (Section IV-B) in a BO context, ending with a final overview (Section IV-C).

A. Gaussian Processes

In BO settings, the *Gaussian process* (GP) [34] is the preferred choice for the prior distribution, or surrogate model, for $f(\cdot)$. For any $x \in \mathcal{A}$, this prior assigns to each value of $f(x)$ a Gaussian probability distribution which depends on x:

$$f(x) \sim \pi_x(\cdot) = \mathcal{N}(\mu_0(x), \sigma_0^2(x, x)). \tag{2}$$

In equation (2), $\mu_0(\cdot)$ and $\sigma_0^2(\cdot,\cdot)$ denote the mean and kernel functions, respectively, and they constitute the GP model hyperparameters. These functions serve as "initial guesses" on values of $f(\cdot)$ and its uncertainty, a starting point for the BO algorithm, which will update them with observed values. A constant mean function $\mu_0(\cdot) \equiv \mu_0$ is often adopted, whereas the choice of the kernel is more delicate since it influences the smoothness of the process. Common choices include the squared exponential or Radial Basis Function (RBF) kernel and the MatÃ©rn kernel [34]. In this work, we assume $\mu_0(\cdot) \equiv \mu_0$, and we use the MatÃ©rn kernel [7], parametrized by the smoothness parameter ν. The RBF kernel is unsuitable for our case since it gives the GP a substantial degree of smoothness, which is unrealistic in many practical scenarios (see [35]).

We now examine the posterior distributions for these hyperparameters. Let $H = \{(x_1, f(x_1)), \ldots, (x_n, f(x_n))\}$ be the history of n past observations, which we also indicate as H_n when emphasizing its cardinality. Specifically, observation i consists of the configuration vector x_i and the associated evaluation of the objective function $f(x_i)$. Having observed values in H, one can compute the posterior distribution of each $f(x)$, starting from the prior distribution in equation (2) and considering these observations. The posterior is the conditional distribution of $f(x)$ given H_n, which, in this case, is a Gaussian distribution with mean $\mu_n(\cdot)$ and variance $\sigma_n^2(\cdot)$ (see, for instance, [7]):

$$f(x)|H_n \sim \pi_x(\cdot|H_n) = \mathcal{N}(\mu_n(x), \sigma_n^2(x)). \tag{3}$$

Note the conditioning symbol $|$ in equation (3). We can compute the posterior mean and variance in closed form by well-known properties of GPs [7]:

$$\begin{aligned} \mu_n(x) = \mu_0(x) + \sigma_0^2(x, x_{1:n})^T \, \sigma_0^2(x_{1:n}, x_{1:n})^{-1} \cdot \\ \cdot \left(f(x_{1:n}) - \mu_0(x_{1:n})\right), \end{aligned} \tag{4}$$

$$\sigma_n^2(x) = \sigma_0^2(x,x) - \sigma_0^2(x, x_{1:n})^T \, \sigma_0^2(x_{1:n}, x_{1:n})^{-1} \cdot \\ \cdot \, \sigma_0^2(x, x_{1:n}). \tag{5}$$

In equations (4) and (5), $\sigma_0^2(x, x_{1:n})$ indicates the column vector of values of the $\sigma_0^2(\cdot,\cdot)$ function applied to pairs $(x, x_1), \ldots, (x, x_n)$, and similarly for $f(x_{1:n})$ and $\mu_0(x_{1:n})$. Analogously, $\sigma_0^2(x_{1:n}, x_{1:n})$ is the matrix of values of $\sigma_0^2(x_i, x_j)$ with $i, j = 1, \ldots, n$. Given a configuration x, after n evaluations, this probabilistic model allows us to attribute both the current pointwise estimate of $f(x)$ and a measure of uncertainty on such estimate, represented by the posterior mean $\mu_n(x)$ and variance $\sigma_n^2(x)$.

B. Acquisition Function

BO formulates a proxy problem at each step – the maximization of the *acquisition function*. This function depends on the history and on the GP model at the current algorithm iteration n and measures the utility of evaluating the objective function $f(x)$ in a given configuration x. Formally, we denote this as $a(x|H_n) : \mathcal{A} \to \mathbb{R}$, or $a(x)$ for short. The BO iterative algorithm optimizes this function at each round instead of directly optimizing the objective function itself.

Since the acquisition function is a measure of expected utility, it has larger values in points in which the algorithm should evaluate the objective function to obtain the most information about the optimum location. This means that the acquisition function must strike a delicate balance – the exploration-exploitation trade-off. On one side, we have points with large uncertainty because they lie in a not yet explored domain region. Choosing such points to evaluate the objective $f(\cdot)$ is appealing, especially early on in the optimization procedure, because it would enable a massive decrease in the overall uncertainty, i.e., an increase in the amount of available information about the optimum. On the other hand, the algorithm seeks to find the optimum of the objective function; therefore, it should also choose to evaluate points that most likely (according to the GP model) give small values of $f(\cdot)$. This means exploiting the already available information on the location of the optimum, especially in the late iterations of the algorithm.

This work considers the Expected Improvement (EI) acquisition function. The EI over the best value f_n^* found by the optimization process so far is:

$$EI_n(x) := \mathbb{E}_{\pi_x(\cdot|H_n)}[\max(f_n^* - f(x), 0)] \\ \text{with } \; f_n^* = \min_{i \le n} f(x_i). \tag{6}$$

The expectation is taken under the current posterior distribution $\pi(\,\cdot\,|H_n)$ of $f(x)$, given history H_n. Equation (6) means that we maximize the expected value of the improvement over the current best point f_n^*, based on information collected so far (i.e., the points in the history H_n).

We use the EI acquisition function because it is the most widely used within the BO literature. The proposed approach is generally compatible with any other acquisition function, such as the Lower Confidence Bound [7], [35].

Algorithm 1: Generic Bayesian Optimization Algorithm.

```
1: choose $n_0$ initial points
2: evaluate $f(\cdot)$ in the initial points, add all evaluations to
   history $H$
3: for iterations $n = 1 : N$ do
4:     update the current posterior distribution of the GP
       model with data in $H$
5:     find point $x_{n+1}$ which maximizes the acquisition
       function $a(\cdot)$ under the current model
6:     evaluate $f(x_{n+1})$, add performed evaluation to $H$
7: end for
8: return estimated optimum $\widehat{x}$
```

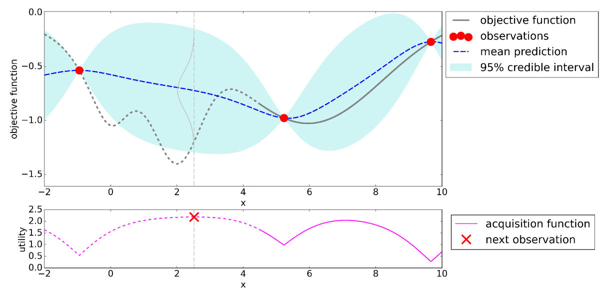
In constrained optimization scenarios, the acquisition function also considers constraints. In particular, the literature has proposed [9] a generalization of EI to the constrained setting. It is the *Expected Improvement with Constraints* (EIC) acquisition function, which accounts for the probability of a point satisfying the constraints:

$$EIC_n(x) := EI_n(x) \cdot \mathbb{P}_{\pi_x(\cdot|H_n)}\left(g(x) \in [G_{\min}, G_{\max}]\right). \tag{7}$$

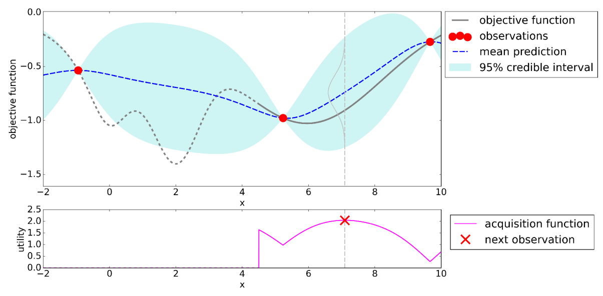
This method requires a probabilistic model for the constraint, e.g., a GP on the values of $g(x)$. This GP can be either independent of the one placed on $f(x)$, or derived from it (like in CherryPick, see Section VI-A1). Note that a constraint-aware acquisition function may only discourage the selection of unfeasible points rather than straight up forbidding it: this is the case for the EIC presented in equation (7).

C. Summary of Bayesian Optimization

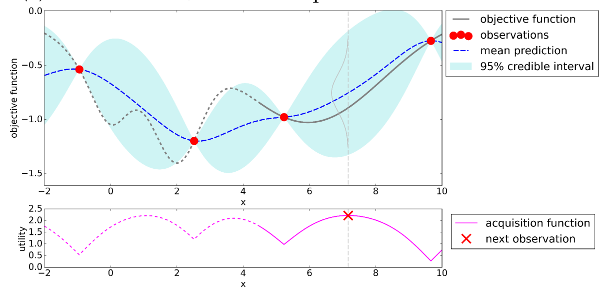
Algorithm 1 summarizes the core procedure for a BO algorithm (both constrained and unconstrained), while Fig. 1(a) and (b) present a visual summary of how BO works. In Fig. 1(a), the gray line represents the actual objective function $f(\cdot)$ to minimize under a constraint. We represent the unfeasible portion of the optimization domain by using a dashed line for the corresponding part of the objective function. Since the constraint has a black-box nature, this unfeasible portion is unknown. In Fig. 1(a), we also plot the GP estimates of $f(\cdot)$ in terms of its mean function (dashed blue line) and 95% credible intervals (light blue area). The three red dots represent sampled points, which have reduced uncertainties compared to other domain points (or zero uncertainty if not accounting for noise in observations of $f(\cdot)$). As previously explained, the GP model attaches a Gaussian probabilistic estimate to $f(x)$ for each x. In Fig. 1(a), the Gaussian curve in light gray represents this estimate when $x = 2.5$. In the bottom part of the figure, we plot the values of the acquisition function for each value of x. We will evaluate the crossed red point in the next round since it has the largest expected utility among all points in the domain. Fig. 1(b) represents the subsequent BO iteration after the new point evaluation and the computation of a new maximizer of the acquisition function.



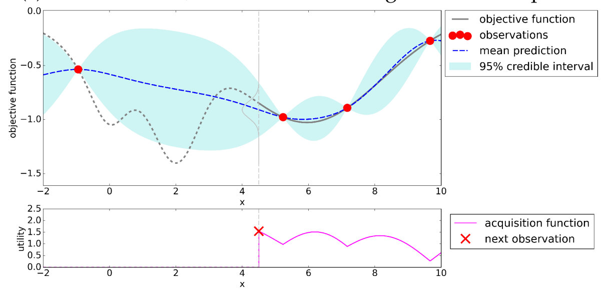
(a) Iteration three, the crossed point will be evaluated next.



(c) Iteration three, the ML model changes the chosen point.



(b) Iteration four, after the evaluation and the GP update.



(d) Iteration four, we have avoided an unfeasible point.

Fig. 1. Illustrative examples of (constrained) Bayesian Optimization: classical (1a, 1b) and ML-integrated version (1c, 1d).

V. Encoding Information From Machine Learning in Bayesian Optimization

Our main contribution is the integration of ML techniques into the BO framework. ML methods can be helpful in optimization problems because of their predicting capabilities. This is especially true in a setting where information on the objective $f(\cdot)$ and the constraint $g(\cdot)$ is scarce, given that both of them are black-box, expensive-to-evaluate functions. For instance, it is possible to convey information about the violation of constraints by guiding the search towards points $x \in \mathcal{A}$ that most likely satisfy them. The BO algorithm benefits from this approach, as our goal is ultimately to find optimal configurations that are also feasible, i.e., points x such that $g(x) \in [G_{\min}, G_{\max}]$ (see equation (1)). Unfeasible points represent a waste of resources and computational time in a recurring job setting, providing additional unnecessary costs.

ML models provide benefits especially considering their accuracy in predicting the execution times of the applications of interest. Besides the promising results on the matter in the literature (see [10], [11], [12], [13]), our preliminary analysis shows good prediction capabilities on the target applications, as we mention in Section VI-C. The idea of BO incorporating information independent of its GP model was first considered in [36].

A. Modifying the Acquisition Function

The core contribution of MALIBOO is integrating the BO acquisition function with information from ML models. The acquisition function is the centerpiece of the BO algorithm, as it single-handedly drives the optimization process by selecting new configurations and managing the exploration-exploitation trade-off. Thus, it is a crucial element for improving a BO algorithm.

Let $\widehat{g}(x) : \mathcal{A} \to \mathbb{R}$ be a predicting function for $g(x)$, that is, a function which outputs a prediction on the constrained resource from equation (1) given configuration x. In our case, $\widehat{g}(\cdot)$ is the estimate of $g(\cdot)$ from an ML regression model, which uses data $\{x_1, \dots, x_n\}$ collected up until the current BO iteration n, and that we train in an online fashion. From a formal perspective, we can characterize an ML-integrated acquisition function as follows:

$$\widetilde{a}(x) = F(a(x), \widehat{g}(x)). \tag{8}$$

The $F(\cdot,\cdot) : \mathcal{A}^2 \to \mathbb{R}$ function in equation (8) encodes the relation between the original acquisition function $a(\cdot)$ and the ML model $\widehat{g}(\cdot)$. We provide two examples of such relation:

- $\widetilde{a}_1(x) = a(x) \cdot \exp(-k\,\widehat{g}(x))$, the latter term being a [0,1]-valued weight called *nascent minima distribution function* [37] of degree k. Through this exponential term, $\widehat{g}(x)$ becomes an acquisition-like function of its own. That is, it has values closer to 1 if the predicted value of the constrained resource $\widehat{g}(x)$ is small and closer to 0 if such prediction is large. This solution handles scenarios where a lower value of the constrained resource is desirable, like for execution time or bandwidth. Instead, if we aim for a higher value, it is possible to use the similarly defined coefficient $\widetilde{a}_1(x) = a(x) \cdot [1 - \exp(-k\,\widehat{g}(x))]$;
- $\widetilde{a}_2(x) = a(x) \cdot I\{\widehat{g}(x) \in [G_{\min}, G_{\max}]\}$, with I being the indicator function. In this case, the acquisition function is zero in areas where we predict that $g(x)$ violates the constraints (i.e., we exclude these areas from the search). We, therefore, use the ML model $\widehat{g}(x)$ to approximate the feasible domain at the current iteration.

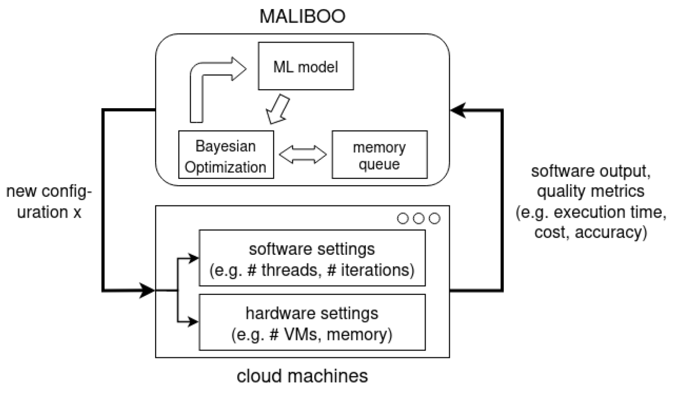

MALIBOO
ML model
Bayesian Optimization
memory queue
new configuration x
software output, quality metrics (e.g. execution time, cost, accuracy)
software settings (e.g. # threads, # iterations)
hardware settings (e.g. # VMs, memory)
cloud machines

Fig. 2. MALIBOO architecture.

We can also combine these factors, creating an ML-integrated acquisition function that accounts for the magnitude of values and feasibility. Since the ML model carries information about the violation of constraints, the new acquisition function is likely to consider fewer unfeasible configurations, improving the algorithm performance.

The proposed algorithm is agnostic to the choice of the original acquisition function $a(\cdot)$, predicting function $\widehat{g}(\cdot)$, and encoding function $F(\cdot,\cdot)$. In particular, we may adopt any other ML model for $\widehat{g}(\cdot)$, ranging from Random Forests to XGBoost, which are commonly used for performance prediction in ICT domains [11], [38].

Introducing an ML model may sometimes lead to the acquisition function becoming non-convex or no longer having a closed-form expression. In this case, we rely on traditional non-convex optimization techniques, such as Nelder-Mead heuristics [39], to find the maximum of the acquisition function.

We represent the potential impact of ML integration and the usage of the $\widetilde{a}_2(x)$ acquisition function in Fig. 1(c) and (d). They depict a similar example as Fig. 1(a) and 1(b), but this time we take the feasibility of the points into account. Indeed, if the ML model can detect the left part of the optimization domain as unfeasible, then it will set the corresponding values of the acquisition function to zero (see the bottom part of Fig. 1(c)). This impacts the choice of the BO algorithm, which in this example chooses a feasible point instead of an unfeasible one (unlike in Fig. 1(a)). In this way, we prevent the exploration of an unfeasible configuration.

B. Proposed Approach

Fig. 2 offers an overview of the implementation architecture of MALIBOO to optimize cloud configuration. Furthermore, we summarize the procedure we propose to integrate BO and ML in Algorithm 2. MALIBOO adopts a first-in-first-out memory queue for discrete features to prevent exploration of already visited values, as suggested by the taboo search meta-heuristic methods [40]. In this memory queue, we save the last q configuration vectors visited by the algorithm. We exclude configurations currently in the queue from being selected again until they have shifted out, i.e., after q iterations.

Algorithm 2: MALIBOO Algorithm.

```
 1: choose $n_0$ initial points
 2: evaluate $f(\cdot)$ in the initial points, add all evaluations to
    history $H$
 3: for iterations $n = 1 : N$ do
 4:     update the current posterior distribution of the GP
        model with data in $H$
 5:     train model $\widehat{g}(\cdot)$ with data in $H$
 6:     find point $x_{n+1}$ which maximizes the acquisition
        function $\widetilde{a}(\cdot)$ under the current model
 7:     evaluate $f(x_{n+1})$, add performed evaluation to $H$
 8:     update memory queue with $x_{n+1}$
 9:     if stopping criteria (if any) are met then
10:         terminate the algorithm
11:     end if
12: end for
13: return estimated optimum $\widehat{x} = \text{argmin}_{x \in H} f(x)$
```

Similarly to regular BO, at each round, we maximize (see algorithm 2, step 6) one of the acquisition functions presented in Section V-A, which incorporates the ML model trained at step 5. Then, we evaluate the new configuration (step 7) as usual, and we update the memory queue (step 8). Finally, we may employ a termination criterion (steps 9-11) for our algorithm based on the observed values of the constrained resource $g(\cdot)$ and its relation to its bounds $G_{\min}$ and $G_{\max}$. The presence of this criterion and its specific implementation are heavily dependent on the optimization problem and application under scrutiny, since values of $g(\cdot)$ have different significance in each scenario. Suppose, for instance, that there is a contrasting relationship between $g(\cdot)$ and the target function $f(\cdot)$, like in the case of minimizing a budget (money, fuel, etc.) while meeting a time threshold. In this case, using more resources (i.e., having larger values of $f(\cdot)$) results in a lower execution time – meaning that a time that is just under the threshold likely consumes the least amount of resources for that configuration to be feasible. In such a scenario, we would choose to employ a termination criterion, and we could design it as follows: the algorithm continues until the evaluated running time at the current iteration is sufficiently close to the time threshold, i.e., $g(x_n) \in [\alpha\, G_{\max},\ G_{\max}]$, with $\alpha \in (0, 1)$. In case we use a stopping criterion, after termination, we have likely found the optimal configuration or at least a near-optimal one. Therefore, we execute all subsequent runs using such a configuration. On the contrary, we would not use any termination criteria in unconstrained optimization scenarios, or any in which the relation between the objective and the constrained resource is unclear. In these cases, the MALIBOO algorithm continues until it reaches the prescribed maximum number of iterations, similar to regular BO.

VI. Experimental Settings

In this section, we present our experimental setup for validating MALIBOO. In Section VI-A, we describe the approaches on which we compare our algorithm with the literature. In Section VI-B, we describe the specific applications we consider and the algorithm parameters we use in Section VI-C. The results themselves are instead detailed in Section VII.

A. Reference Comparison Approaches

We now describe the different methods we compare with MALIBOO: CherryPick, SVM-CBO, and OpenTuner. They represent a varied range of approaches: in particular, they are the reference BO approach for cloud systems, a hybrid BO-ML approach, and a widely used non-BO black-box autotuner, respectively.

1) Cherrypick: CherryPick [3] is one of our reference approaches for constrained BO-based cloud optimization, as well as one of the seminal works on cloud systems optimization proposed in the literature that handles the constrained case. Hence, we consider the model used in CherryPick as a competitor for several of our validation experiments. In this case, the goal is to find the optimal configuration of a Spark application running in a cloud system to minimize its execution cost. The model used by CherryPick is the following:

$$\min_{x \in \mathcal{A}} P(x)T(x) + \varepsilon \quad \text{s.t.} \quad T(x) \leq T_{\max}, \tag{9}$$

with $P(x)$, $T(x) > 0 \, \forall x \in \mathcal{A}$. In particular, $f(x) = P(x)T(x)$, where $P(x)$ is the variable cost per unit of time $T(x)$ given configuration x. This cost model is realistic in many cloud computing scenarios, where users pay, e.g., for the number of parallel cores or VMs they wish to use. CherryPick also assumes the deterministic price function $P(x)$ to be proportional to the number of VMs or cores used by the application job, which is always included in the configuration vector x. In our general model in equation (1), this corresponds to setting the constraint function $g(\cdot) \equiv T(\cdot)$ and the bounds $G_{\min} = 0$ and $G_{\max} = T_{\max}$. In this way CherryPick introduces a relation between $f(\cdot)$ and $g(\cdot)$, an assumption which our general model does not make.

In particular, CherryPick considers the *Expected Improvement with Constraints* (EIC) acquisition function presented in Section IV-B:

$$\begin{aligned} EIC_n(x) &:= EI_n(x) \cdot \mathbb{P}_{\pi_x(\cdot|H_n)}\left(T(x) \leq T_{\max}\right) \\ &= EI_n(x) \cdot \mathbb{P}_{\pi_x(\cdot|H_n)}\left(f(x) \leq P(x)\, T_{\max}\right). \end{aligned} \tag{10}$$

In the last equality, the constraint $T(x) \leq T_{\max}$ is expressed as a function of $f(x) = P(x)T(x)$ since, in BO methods, the GP prior is placed on the target function $f(\cdot)$, not on $T(\cdot)$. For our comparison, based on the $\widetilde{a}_1(\cdot)$ and $\widetilde{a}_2(\cdot)$ described in Section V-A, we develop several variants of the acquisition function:

- $g_A(x) = EIC(x)$ (or $EI(x)$ in unconstrained scenarios), the original acquisition function, which we use as baseline;
- $g_B(x) = g_A(x) \cdot \exp(-k\,\widehat{T}(x))$, with the nascent minima distribution function coefficient;
- $g_C(x) = g_A(x) \cdot I\{\widehat{T}(x) \leq T_{\max}\}$, with the indicator function;
- $g_D(x) = g_A(x) \cdot \exp(-k\,\widehat{T}(x)) \cdot I\{\widehat{T}(x) \leq T_{\max}\}$, the combination of cases B and C.

Because CherryPick is limited to constrained minimization, we cannot use the same acquisition functions in the unconstrained case. In such scenarios, we can use a similar acquisition function using regular EI: $g_B(x) = EI(x) \cdot \exp(-k\,\widehat{f}(x))$, namely replacing $\widehat{T}(x)$ with $\widehat{f}(x)$ in the exponential component. This acquisition function guides the optimization process towards points the ML model deems promising, e.g., with small values of $f(\cdot)$.

2) Svm-Cbo: The second competing approach we focus on is SVM-CBO [32] since it is one of the algorithms from the literature combining BO and ML models, and hence one of the most comparable approaches to ours. In this work, authors attempt to find the hyperparameter set that maximizes the accuracy score of a DNN while subject to black-box deployability constraints.

As anticipated in Section II, the SVM-CBO algorithm consists of two phases. In phase 1, starting from a pool of evenly distributed points, it initializes a Support Vector Machine (SVM) classification model. This model aims to approximate the feasible domain of the problem, by distinguishing between feasible (i.e., the DNN is deployable on a memory-constrained device) and unfeasible configurations. The algorithm then iteratively collects new configurations using a non-BO-based acquisition method that considers the SVM separation surface and a measure of the search space coverage. The DNN is compiled on the target device at each new configuration, and the algorithm updates the SVM model with the feasibility information of such configuration. This phase ends after a fixed number of iterations.

In phase 2, the algorithm performs pure BO to find the DNN configuration, which maximizes the accuracy score. In this phase, the algorithm restricts the optimization domain to the approximated domain from phase 1. Again, the BO algorithm continues until the fixed iteration budget runs out.

Similarly to MALIBOO, this approach exploits ML to estimate the feasible domain. However, in SVM-CBO, this estimation takes place separately from the optimization rather than jointly. For this reason, MALIBOO can exploit the iteration budget better than SVM-CBO. Our experimental results in Section VII support this claim.

3) Opentuner: Finally, we compare against OpenTuner [15], a popular open-source autotuner. It uses an ensemble of multiple search techniques to conduct black-box constrained or unconstrained optimization of the given application. A meta-search algorithm guides the exploration process, allocating more tests to techniques which perform well. The techniques coordinated by the meta-algorithm include classical optimization methods such as differential evolution and greedy mutation variants. Individual techniques share results through a common database, so that improvements made by one of them can also benefit the others. This sharing occurs in technique-specific ways; for example, evolutionary techniques add results found by other techniques as members of their population. The default meta-algorithm in OpenTuner is the “multi-armed bandit with sliding window, area under the curve credit assignment”, also known as AUC Bandit. The ensemble of several different methods allows extensive exploration of the optimization domain, while the clever allocation of tests by the meta-technique encourages the exploitation of promising paths.

TABLE I
SUMMARY OF OPTIMIZATION PROBLEMS

Reference application	Domain dimension	Optimization
Spark	1 or 2	min. cost, constr. on time
Stereomatch	4	min. cost, constr. on time
GPU benchmarks	4	min. cost (unconstrained)
LiGen	8	min. f(cost, accuracy), constr. on accuracy

The library includes classes for constrained and unconstrained autotuning, named MinimizeTime and ThresholdAccuracyMinimizeTime, respectively. Despite their name, one can implement derived classes to account for any objective and constraint functions.

B. Experiment Setting

In this context, we test the MALIBOO algorithm in several scenarios, on both public and private cloud systems. The first three scenarios involve the Apache Spark Big Data analytics framework [41]. Other scenarios involve the Stereomatch edge computing application [42] and the BFS and MD GPU benchmarks. Finally, we test our technique on LiGen [43], a real-world scientific computing application.

The applications mentioned above represent several system types commonly employed in the industry. Big data applications often run on cloud servers because they offer easy access to powerful analysis frameworks such as Apache Spark. On the other hand, an edge computing application such as Stereomatch represents an emerging access pattern to cloud resources, in which data collection happens at the IoT layer while processing takes place in the cloud. GPU applications are prevalent in tasks that can use acceleration for parallelizable code, ranging from neural networks [44] to large-scale image processing programs [45]. The usage of scientific computing applications is widespread in most science and engineering areas, from spacecraft trajectory simulation to molecule simulation [46].

We summarize the optimization problems described in the next sections in Table I.

1) Big Data Applications: We now describe the three Apache Spark applications, which we optimize on the number x of parallel cores used to run them (x is therefore one-dimensional):

- *Query26* is an interactive query from the TPC-DS industry benchmark,[1] and represents SQL-like tasks. We execute it with input datasets I of varying sizes, specifically 250 GB, 750 GB, and 1000 GB.
- *Kmeans* is a well-known statistical clustering technique and a typical example of an iterative task. We execute it on Spark-Bench[2] by providing it as input datasets I with 100 features (columns) and varying sizes (rows): 5, 10, 15, and 20 million.
- *SparkDL Transfer Learning*[3] is a Big Data analytic tool that applies transfer learning to deep learning applications by relying on the Spark MLlib for the last layer and on TensorFlow for the featurization part. This work considers an image binary classification task with input datasets I containing 1000, 1500, and 2500 images.

We performed the Spark experiments on two computing environments: the Microsoft Azure public cloud and a private IBM Power8 cluster. In particular, we executed Query26 and SparkDL on Microsoft Azure using the HDInsight service with workers based on 6 D13v2 VMs, each with 8 CPU cores and 56 GB of memory. Additional Query26 experiments involve up to 26 VMs of 5 different flavors provided by Azure, each with 2 to 16 CPU cores and 2 to 90 GB of memory, each characterized by a different memory size. We ran Kmeans on an IBM Power8 deployment, including 4 VMs, each with 12 cores and 58 GB of RAM, for a total of 48 CPU cores available for Spark workers, plus a master node with 4 cores and 48 GB of RAM. These two systems are representatives of different computing environments. The Microsoft Azure public cloud can suffer from resource contention, and application execution times might experience more variability. On the other hand, the private IBM Power8 cluster is fully dedicated to our experiments without any other concurrent activity, i.e., with no resource contention.

For all three Spark applications, we perform one separate experiment for each input dataset I, for a total of 10 experiments. We also carry out one *extrapolation* experiment with additional data for each of the three applications. In each of these experiments, for the largest input dataset I available for each application (1000 GB, 20 million rows, and 2500 images, respectively), we give additional profiling data to the ML model $\widehat{T}(\cdot)$ estimating the performance (see Sections V-A and VI-A). These additional profiling data consist of all previous runs with smaller input datasets I (250-750 GB, 5-10-15 million rows, and 1000–1500 images, respectively), which we assume are available from previous application runs and are used in the training phase of the regression model $\widehat{T}(\cdot)$, in addition to the points visited by the BO algorithm. This kind of experiment is tailored for Big Data settings, which often require running multiple data analysis tasks or prediction models, on input datasets with increasing size. The goal of such experiments is to check whether we can save on exploratory runs with the larger datasets, which are usually much more expensive than the ones with small datasets. Indeed, if the ML model can extrapolate the performance with large input datasets by using data from small ones, the MALIBOO optimization process would be more efficient. According to preliminary analysis in [11], ML models show good extrapolation capabilities on the applications of interest, which motivates us to conduct the extrapolation experiments.

Finally, we show the results of an additional experiment with the Query26 application. This experiment is similar to the previous ones, but we choose both the VM type and the number of VM instances. We therefore have a two-dimensional optimization problem of configuration $x = (x_1, x_2)$, in which $x_1 \in \{2, \dots, 90\}$ represents the amount of memory (in GB) which characterizes the VM (choosing among Azure A3, A4, D12v2, D13v2, and D14v2), and $x_2 \in \{1, \dots, 26\}$ is the number of VMs used.

[1]https://www.tpc.org/tpcds
[2]https://codait.github.io/spark-bench
[3]https://github.com/databricks/spark-deep-learning

2) Edge Computing Application: Another scenario we consider for validation uses *Stereomatch* [42], an image-processing edge computing application. Stereomatch evaluates the disparity value between a pair of stereo images (i.e., coming from the same scene but observed by two cameras), which can then be used to calculate the depth of objects in that scene. This application uses adaptive-shape local support windows for each pixel based on color similarity. In this case, $x = (x_1, x_2, x_3, x_4) \in \mathbb{N}^4$ consists of four independent parameters which influence its execution time: x_1 is the number of parallel threads, x_2 is the color similarity confidence, x_3 is the granularity of the disparity hypotheses to test, and x_4 is the support window arm length. This parameter space is much larger and higher-dimensional than in the previous experiments. We execute this application with a fixed input dataset I containing 40 pairs of images. Since we use a single input dataset for Stereomatch, we perform a single experiment with it.

3) GPU-Based Benchmarks: BFS and *MD*[4] are two GPU benchmark applications from the HPC community. The former implements Breadth-First Search on a tree-like structure, while the latter is a Molecule Dynamics simulator. They both have a four-dimensional parameter space. These parameters are the block size $x_1 \in \mathbb{N}$ used for launching the application kernel, the amount $x_2 \in \mathbb{N}$ of work per thread, and two other algorithmic-specific knobs related to texture memory ($x_3 \in \mathbb{N}$) and floating-point precision ($x_4 \in \{32, 64\}$). For each application, we perform one experiment of unconstrained optimization.

4) Molecular Docking Application: LiGen is a molecular docking application part of the EXSCALATE drug discovery platform [43]. This framework runs in a scientific computing private cloud environment and conducts extreme-scale virtual screening campaigns to quickly obtain information for the drug discovery pipeline. Indeed, the reduction of the time required to find a therapeutic cure is of vital importance, as seen in the outbreak of the COVID-19 pandemic. Within the EXSCALATE platform, the LiGen code simulates the multiple ligand-pocket interactions, finding promising *docking* poses of the ligand within the pocket through multiple restarts of a gradient descent algorithm. After a clustering analysis, the application chooses several representative poses and evaluates them with a *scoring* function [47]. One can validate the quality of the docking solution by measuring the average Root Mean Square Distance (RMSD) for 100 ligand-proteins pairs with a known optimal crystal position.

Despite this application exposing several software knobs, in this experimental campaign, we selected the eight most significant ones. Six of these parameters influence the accuracy of the docking algorithm and, consequently, its performance, while the other two only influence the performance. In particular, in the former category, we have the number of gradient descent restarts $x_1 \in \{1, \ldots, 5\}$ and three other quantities (x_2, x_3, x_4) controlling the degree of thoroughness of the process. The other two accuracy-related parameters are the clustering distance $x_5 \in \{1, \ldots, 4\}$ of the algorithm and the number $x_6 \in \mathbb{N}$ of chosen poses to evaluate. The last two parameters that only influence the performance are the number $x_7 \in \mathbb{N}$ of CUDA threads in each block and the reading buffer size $x_8 \in \mathbb{N}$. The tuning of these parameters is critical to obtaining a better performance-accuracy trade-off, but the total number of configurations in this parameter space amounts to over 60 million, which mandates the need for a sensible tuning approach.

5) Applications Hardware Configuration: We execute all four non-Spark-based applications, i.e., Stereomatch, BFS, MD, and LiGen, in a VM on a private 32-core server with 64 GB of memory and Ubuntu 20.04. The underlying physical node of this server has two AMD EPYC 7282 processors, with 16 cores and 32 threads, and a clock speed of 2.8 GHz. It also has two NVIDIA A100 GPUs with 40 GB of memory each.

We performed one experiment for each of the four non-Spark applications. For each application, we also show average metrics computed over several runs of the algorithm. In particular, for Spark-based applications, we average over executions with 10 different time thresholds and different input data sizes (as described in Section VI-B1); in Stereomatch (which has a fixed input dataset) only over 10 time thresholds; for GPU benchmarks, we average over executions with 10 different random sets of initial configurations; lastly, for LiGen we average over both 10 thresholds and 10 sets of initial points.

C. Algorithm Settings and Comparison

We use the applications described in the previous section to compare all MALIBOO variants (B-D, see Section VI-A) against CherryPick pure BO (represented by variant A), SVM-CBO, and OpenTuner. For SVM-CBO we rely on an adaptation of the original authors' code,[5] while for OpenTuner we run the original library after implementing appropriate objective function objects.

The algorithms run on an Ubuntu 20.04 machine with 16 GB RAM for 30 or 60 maximum iterations for each experiment, depending on the variability in the application performance and the search space size. Besides the execution time of the job, the computation time for a single MALIBOO iteration is about 1.5 seconds, with a maximum of 5 seconds for late iterations (for larger ML models), and up to 10 seconds for extrapolation experiments.

For a fair comparison, in each experiment, we use the same $n_0 = 3$ initial configurations (or $n_0 = 11$ in the LiGen case due to the larger search space) for all algorithms. Moreover, in the constrained scenarios, we choose a grid of 10 evenly spaced resource thresholds and repeat the experiments for each of them. This process represents an increasingly difficult optimization problem as the threshold decreases since the feasible domain keeps shrinking. For SVM-CBO, we keep the same initial points and number of iterations as variants A-D, and we split the total iteration budget with a similar proportion to the one recommended by the authors in [32] (namely, about 10% of initial points, 60% for optimization phase 1, and 30% for phase 2). Finally, to compare with OpenTuner we used its default signature meta-technique, the AUC Bandit coordinating the

[4]https://github.com/NTNU-HPC-Lab/BAT

[5]https://github.com/ricky151192/SVMCBO

DifferentialEvolutionAlt, UniformGreedyMutation, NormalGreedyMutation, and RandomNelderMead search techniques.

For our MALIBOO algorithm, we use a Ridge linear regression model as the predicting function $\widehat{g}(\cdot)$ described in Section V-A. We use default settings for Ridge, as provided by the Scikit-learn Python library [48]: a regularization penalty of 1, and a solution method based on a Singular Value Decomposition (SVD). For the Stereomatch case only, the default value resulted in poor accuracy, and hence, we chose a penalty of 10 after hyperparameter tuning. This ML model is computationally cheap, widely used in many data analysis scenarios, and performs better in predicting the performance of the applications of interest when compared to other alternative methods (see also [11]). In particular, we measure its predicting accuracy by computing the Mean Absolute Percentage Error (MAPE) of the model:

$$MAPE(\boldsymbol{y}, \widehat{\boldsymbol{y}}) = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y_i - \widehat{y}_i}{y_i}\right|,$$

where $\boldsymbol{y}$ is the vector of true values and $\widehat{\boldsymbol{y}}$ is the vector of predicted values by the ML model. According to our analysis of the full profiling datasets, the test-set MAPE of the chosen model is almost always lower than 9%. Even when just a handful of training points are available, errors are mostly within 20%. As for the GP hyperparameters, we choose a constant mean function and a MatÃ©rn kernel, as described in Section IV, with smoothness parameter $\nu = 5/2$. We set $k = 2$ as the exponential term described in Section V-A) and $q = 5$ as the memory queue length. Furthermore, we employ the termination criterion described in Section V-B in all constrained experiments except LiGen since the target and the constrained resource have a contrasting relation in these experiments. Note that in these cases, the unit time cost $P(x)$ (see equation 9) has a much larger impact on the objective $f(x)$ than the execution time $T(x)$, therefore a small $f(x)$ still generally implies a large $T(x)$, as argued in Section V-B. When using the stopping criterion, we choose $\alpha = 0.9$ as the lower bound parameter.

Finally, for the maximization of the non-convex acquisition functions, we use Nelder-Mead heuristics [39] with multiple starting points, to ensure that the method converges to a good local optimum.

VII. Experimental Results

In this section, we compare our proposed algorithm variants with the presented state-of-the-art techniques, in terms of the number of unfeasible runs and their cumulative costs. Each subsection covers a specific type of application: the three Spark-based applications in Section VII-A, the Stereomatch edge computing application in Section VII-B, the GPU benchmarks in Section VII-C, and the LiGen molecular docking application in Section VII-D.

A. Big Data Applications

We describe the results for the three Big Data applications, which represent the simplest cases of optimization experiments we cover. In particular, Section VII-A1 presents the basic mono-dimensional cases whose goal is to identify the optimal total number of cores, while Section VII-A2 presents the extrapolation case. Section VII-A3 describes an extension to a bi-dimensional case in which we choose both the VM flavor and the number of VM instances to use. We recall that in this case, we minimize cloud costs while subject to a constraint on execution time (see equation 9).

TABLE II
Measured Metrics for Spark Experiments

scenario	var. A	var. B	var. C	var. D	SVM	OT
Query26						
unfeasible runs	11.23	4.34	4.11	4.31	17.16	4.10
ratio of unf. costs	0.36	0.14	0.13	0.14	0.48	0.10
mean feasible cost	1	0.87	0.87	0.88	1.15	1.04
Kmeans						
unfeasible runs	10.52	5.74	5.93	5.96	16.38	11.23
ratio of unf. costs	0.34	0.20	0.21	0.21	0.60	0.43
mean feasible cost	1	1.00	1.01	1.00	1.41	1.31
SparkDL						
unfeasible runs	11.65	5.29	5.14	5.24	16.35	7.96
ratio of unf. costs	0.32	0.16	0.15	0.16	0.21	0.15
mean feasible cost	1	0.83	0.87	0.79	3.82	3.66

1) Total Cores Analysis: Table II summarizes the average results on the Apache Spark Big Data applications described in Section VI-B1 over the varying input data sizes and time thresholds. In particular, we report: (i) the average number of executions that have selected an unfeasible configuration, (ii) the percentage of costs (the $f(x)$ in equation (9)) coming from unfeasible configurations with respect to the total costs over the entire 30 iterations, and (iii) the average cost among feasible configurations, normalized over the cost of variant A (i.e., CherryPick). SVM and OT stand for SVM-CBO and OpenTuner, respectively. For the Query26 application, our algorithm variants reduce the unfeasible executions and the ratio of unfeasible costs two to three times compared to the CherryPick pure BO (variant A). We also reduce the average cost of a feasible configuration. In the three Spark applications, we see an average improvement of 2.2 times in the number of unfeasible runs and twice in the total unfeasible cost. The cost of feasible runs improves by 10% on average, and it is always at least competitive with CherryPick in the worst case. On the other hand, in these experiments, SVM-CBO performs considerably worse than any MALIBOO variant. This may be explained by an over-allocation of resources devoted to learning the feasible domain, for a relatively simple optimization problem. Indeed, SVM-CBO allocates some iterations exclusively to train its SVM model (phase 1) and, therefore, has fewer iterations of BO (phase 2), which is a more effective optimization strategy. The comparison with OpenTuner gives an overall positive result too. OpenTuner shows similar performance as MALIBOO for Query26, but performs much worse with the other two applications, finding up to twice the number of unfeasible configurations in the Kmeans case.

Fig. 3 shows a representative example for a run of all algorithms on the Query26 scenario. The left panel shows the number of cores chosen at each algorithm iteration. The green horizontal line is the true optimum of the constrained optimization problem, which we identified by inspection through an exhaustive application profiling. The vertical, dashed red line indicates the

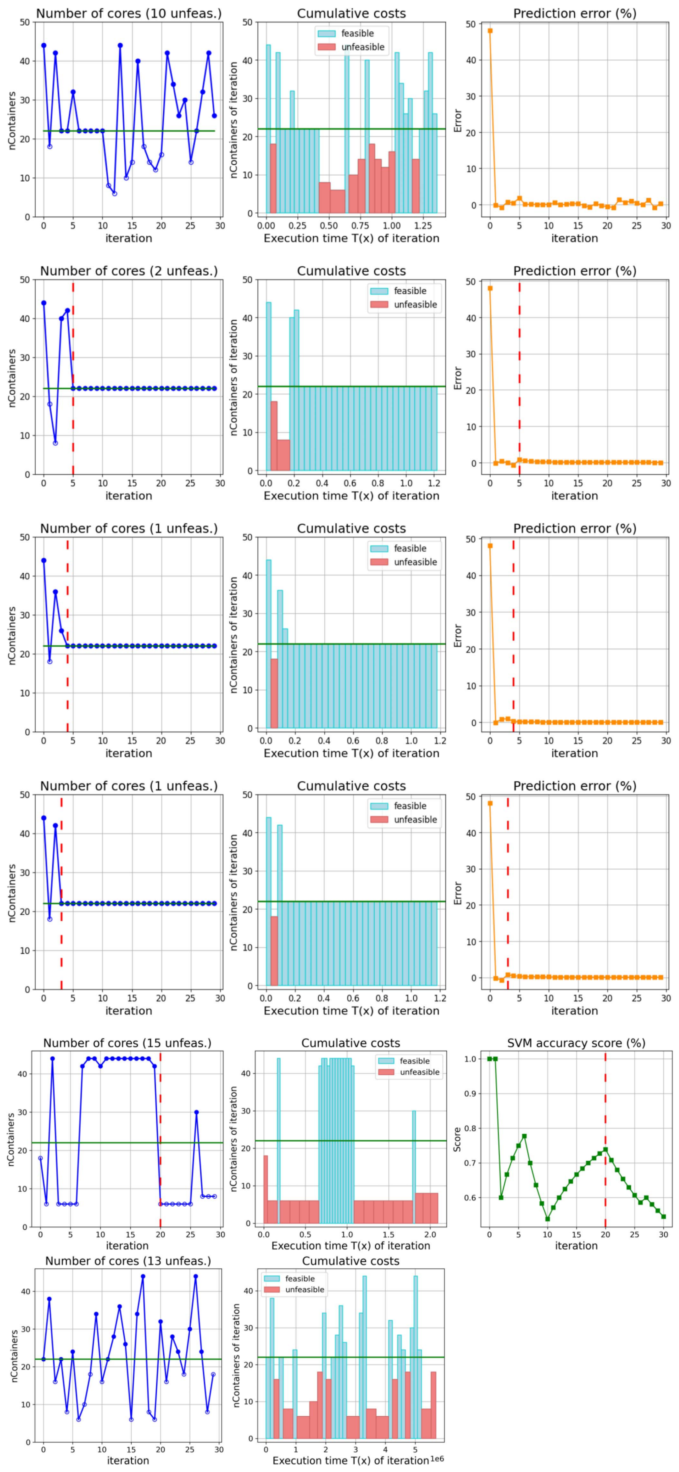


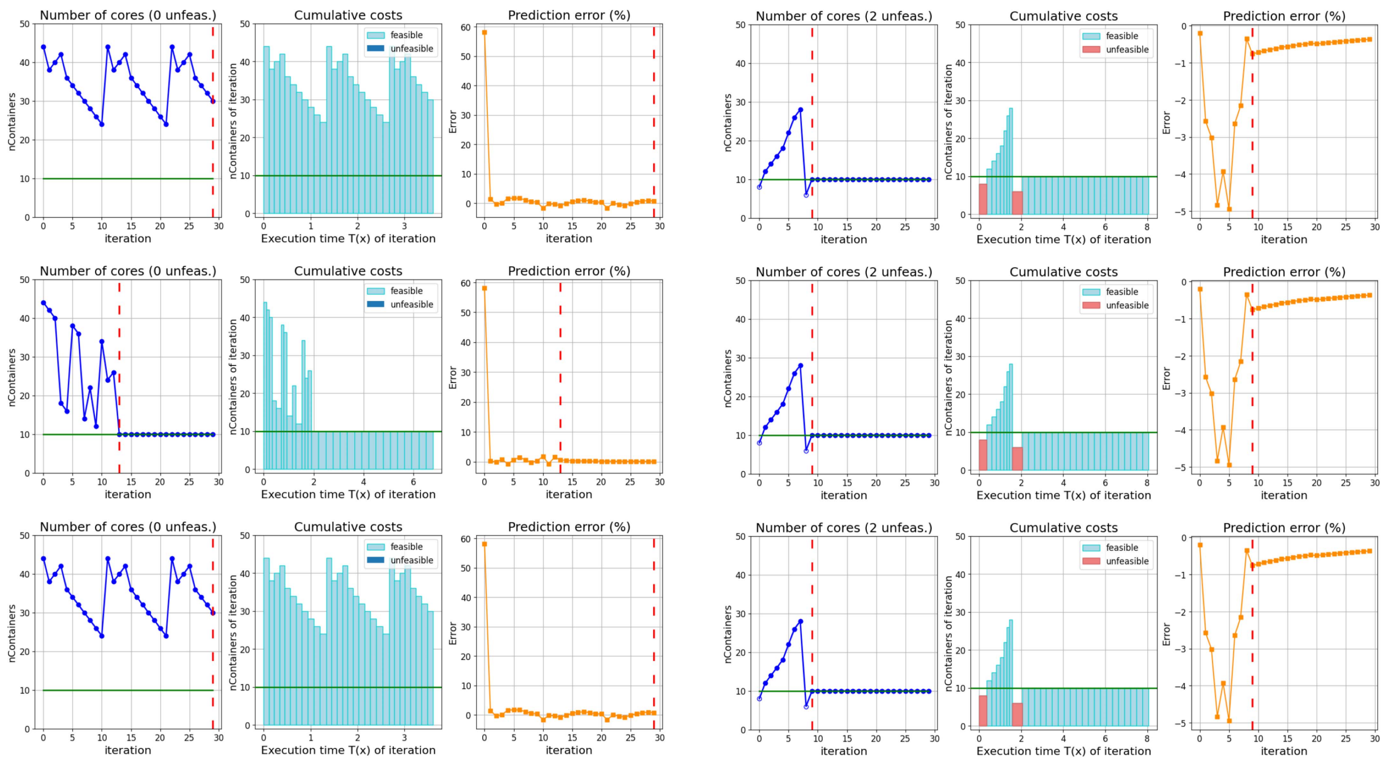
Fig. 3. Spark Query26: comparison of CherryPick pure BO (variant A), MALIBOO variants B, C, and D, SVM-CBO, and OpenTuner, respectively, from top to bottom.

run at which the stopping criterion kicks in and after which we stick to the best configuration found so far by the algorithm (for SVM-CBO in the bottom row, it represents instead the separation between phases 1 and 2). In the center panel, a rectangle represents a single run, with its sides being the execution time of the job (horizontal side) and the number of cores (vertical side). Therefore, the area of a rectangle is proportional to the execution cost $f(x)$ for that particular run (see equation (9)). We highlight in red the rectangles corresponding to unfeasible configurations. The right panel displays the signed percentage errors of the ML model for the execution time, or the model accuracy score in the SVM-CBO case (we do not show any plot for OpenTuner since it does not train any ML models). In particular, at each iteration, MALIBOO performs the training of the model with all data from previous iterations and evaluates the error on the new configuration chosen by the algorithm. Results show that after one initial run with a significant error, our ML models quickly converge to errors close to zero. This initial spike may be explained by the considerable distance between the n_0 initialized points and the first point selected by the algorithm. Indeed, in this particular experiment, the initial points have a small number of cores. Therefore, the algorithm, which is still in the exploration phase, selects a configuration with a large number of cores (as seen in the leftmost spike of the number of cores) because it lies in a region of the domain that is still unexplored. After that, the ML model has enough information to achieve good predicting capabilities, even with few training data points.

As also shown in Table II, from Fig. 3, it is clear that our algorithm drastically reduces the number of unfeasible runs compared to the competing algorithms (from 10-15 to only 1-2) while still achieving the optimum. Overall, our algorithm variants converge to an optimal or feasible near-optimal (i.e., one more or one less than the optimum) number of cores within the given iteration budget over 54% of the times, usually before the 10th execution.

2) Extrapolation Experiments: We also perform extrapolation experiments with the three Apache Spark applications: Query26, Kmeans, and SparkDL. We recall that these experiments correspond to the customary need to run the same Big Data analysis on datasets of increasing size and that, in this case, the ML model receives additional profiling data from smaller datasets in the training phase (see Section VI-B1). In Fig. 4(b), we show a representative extrapolation result for Query26, while Fig. 4(a) reports the corresponding case with the regular algorithm, i.e., without feeding the ML models with additional training data. We do not report SVM-CBO or OpenTuner because it is not possible to perform extrapolation experiments with them.

Table III collects the results of all runs averaged on the varying time thresholds of these extrapolation experiments. In general, the algorithm with additional data has a more aggressive search behavior, trading off a small number of unfeasible runs (usually no more than 2-3) for a lower average cost of individual runs (6 to 26%). A possible explanation is the increased accuracy of the ML model, as seen, for example, in the rightmost plots of Fig. 4(b). The model guides the domain exploration of BO towards more promising points, either because of their lower predicted execution time (variant B), their predicted feasibility (variant C), or both (variant D). Recall that this is a case in which the objective function and the constrained resource are in contrast with one another (see Section V-B). These new points



(a) Regular algorithm without additional data.

(b) Extrapolation algorithm with additional data.

Fig. 4. Query26: comparison between regular (4a) and extrapolation (4b) experiments. From top to bottom, each row represents variants B, C, and D, respectively.

TABLE III
MEASURED METRICS FOR EXTRAPOLATION EXPERIMENTS

scenario	var. A	var. B	var. C	var. D
Query26-extrapolation				
unfeasible runs	3.57	4.64	4.64	4.86
ratio of unf. costs	0.11	0.15	0.15	0.15
mean feasible cost	1	0.83	0.83	0.83
Kmeans-extrapolation				
unfeasible runs	10.00	5.00	5.00	5.00
ratio of unf. costs	0.33	0.17	0.17	0.17
mean feasible cost	1	1.33	1.40	1.32
SparkDL-extrapolation				
unfeasible runs	12.75	8.62	6.29	5.57
ratio of unf. costs	0.35	0.27	0.19	0.16
mean feasible cost	1	0.98	0.83	1.13

TABLE IV
MEASURED METRICS FOR VM FLAVOR EXPERIMENTS

scenario	var. A	var. B	var. C	var. D	SVM	OT
Query26-extended						
unfeasible runs	27.2	20.18	27.08	26.32	20.16	17.16
ratio of unf. costs	0.90	0.67	0.90	0.88	0.66	0.63
mean feasible cost	1	1.009	1.004	1.001	0.998	1.011

are thus more likely to be on the boundary of the feasibility region of the optimization problem, where we have a higher risk of stepping out of the feasibility region. However, these boundary points also give smaller costs (i.e., objective function values). Therefore, the extrapolation version of MALIBOO is best suited for a user with a lower risk aversion.

3) VM Flavor and Number of Instances Analysis: Finally, we perform two-dimensional optimization experiments with the Query26 application, using an input dataset I of size 500 GB. We optimize the execution time on the number of used VMs to run the application and the type of VM. Each VM type has different features, including the type and number of CPUs and the amount of RAM available. Specifically, the RAM uniquely identifies each VM type (see Section VI-B1), and it is the value we include in the vector optimization x (alongside the number of VMs) to encode the choice between different types.

Table IV summarizes the results of the experiments, averaging over the different time thresholds used. We can see that MALIBOO variant B performs much better than CherryPick pure BO (variant A), slashing the number of unfeasible executions and the corresponding costs by about 25%. Variants C and D perform analogously or slightly better than CherryPick, and unlike all other experiments, SVM-CBO shows results comparable to variant B. OpenTuner does achieve slightly fewer unfeasible configurations and costs, although the average cost of feasible executions is larger than all BO-based methods.

We further analyze these results by showing a representative example of a run in Fig. 5. Note that since we no longer deal with a uni-dimensional optimization domain, the scatter plot of the chosen configurations does not accurately depict the distance from the optimum. For this reason, in the center panel, we show instead the percentage of simple regret (i.e., the distance between values of the target function) of the current best solution compared to the true optimum (which is again obtained by

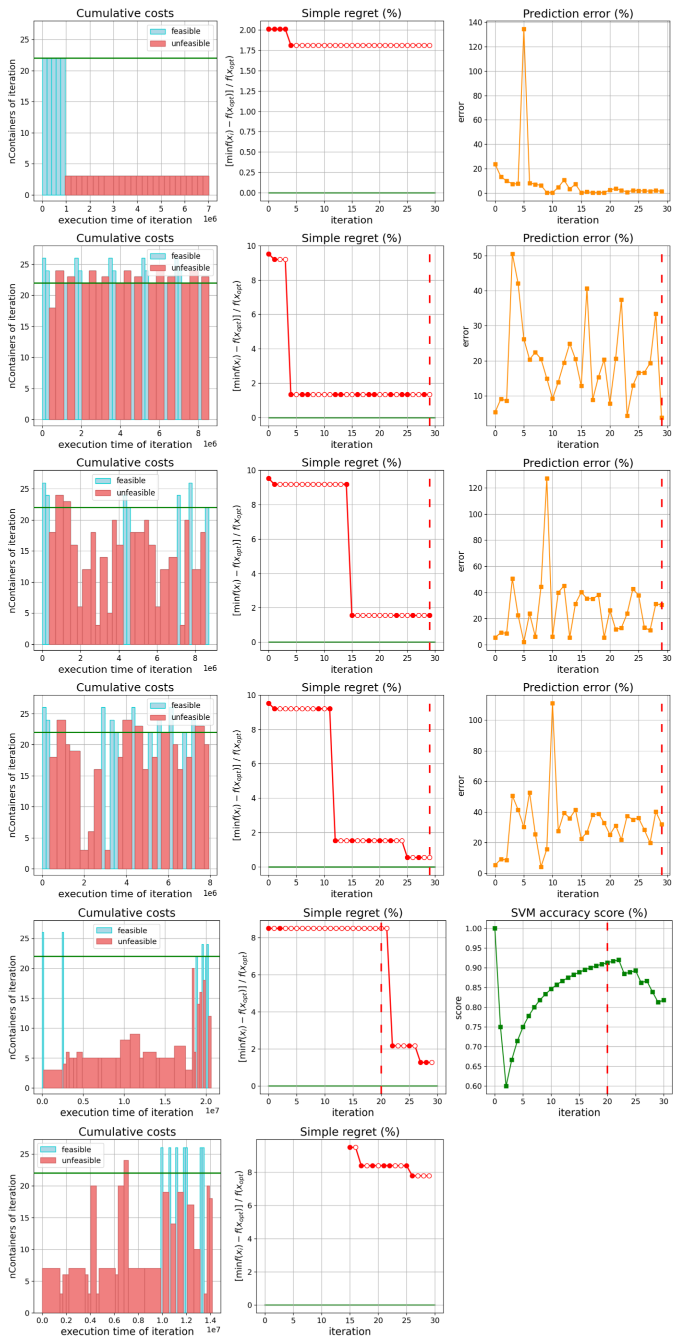


Fig. 5. Query26 extended case: comparison of CherryPick pure BO (variant A), MALIBOO variants B, C, and D, SVM-CBO, and OpenTuner, respectively, from top to bottom.

inspection with an exhaustive search in the parameter space), that is:

$$\widehat{r}_n = \min_{i \leq n} f(x_i) - \min_{x \in \mathcal{A}} f(x) \tag{11}$$

The figure shows that we are dealing with a surprisingly tricky optimization problem despite being less complex and lower-dimensional than the ones in the following sections. In particular, as seen in this and other plots we examined, finding feasible iterations is challenging, and the margin of improvement is

TABLE V
Measured Metrics for Stereomatch Experiments

scenario	var. A	var. B	var. C	var. D	SVM	OT
Stereomatch						
unfeasible runs	28.97	28.29	9.45	8.48	30.05	11.49
ratio of unf. costs	0.48	0.47	0.15	0.14	0.45	0.13
mean feasible cost	1	0.98	0.76	0.90	1.98	1.89

minimal, both in regret and feasible iteration cost. Moreover, all ML models (both in MALIBOO and SVM-CBO) are less accurate in this particular experiment than in most other cases. The MALIBOO models still manage to lower the regret and find feasible configurations. Instead, CherryPick pure BO (variant A) is stuck on an apparent optimum, which is unfeasible and thus cannot improve further on the current best. Also, the comparison with OpenTuner suggests that this particular problem may be less suited for BO-based approaches, instead requiring exploration to a much larger extent.

In conclusion, considering results in Section VII-A1, VII-A2, and VII-A3, our algorithm variants (B-D) outperform CherryPick (here represented by variant A) and SVM-CBO concerning the number of unfeasible runs and the total unfeasible costs (2 to 4 times less), as well as the average cost of feasible configurations (13 to 40% less). MALIBOO also performs better than OpenTuner, or equally as good, in most cases.

B. Edge Computing Application

We now show the results related to the Stereomatch application, in which we again optimize application costs while subject to an execution time constraint. Table V and Fig. 6 show the results for the Stereomatch edge computing application. Similarly to Fig. 5 in the last subsection, we show the bar plot of execution costs, the simple regret, and the prediction errors of the employed ML models for each algorithm iteration. Note that, despite the ML model being less accurate on the Stereomatch application than other workloads, results are still similar to the ones achieved in the other cases. In particular, as summarized in Table V, the improvements of variants B-D compared to variant A are in line with the three Spark scenarios, with a reduction on average feasible costs of about 25%. Once again, we also note that the SVM-CBO algorithm here performs worse than CherryPick or our proposed variants. OpenTuner also performs worse in terms of average feasible cost, and slightly worse in terms of unfeasible executions compared to variants C and D.

C. GPU-Based Benchmarks

In the previous sections, we showed the promising performance of our hybrid BO algorithm on constrained optimization problems in the form of equation (9). We also want to check whether the algorithm brings any benefits in an unconstrained optimization scenario. The BFS and MD applications described in Section VI-B3 are ideally suited for the unconstrained case. Since both of them are GPU-only applications, they do not have the number of cores among the optimization variables, and therefore, we are minimizing their running time directly: $f(x) = T(x)$. In this context, we focus on variant B, which can

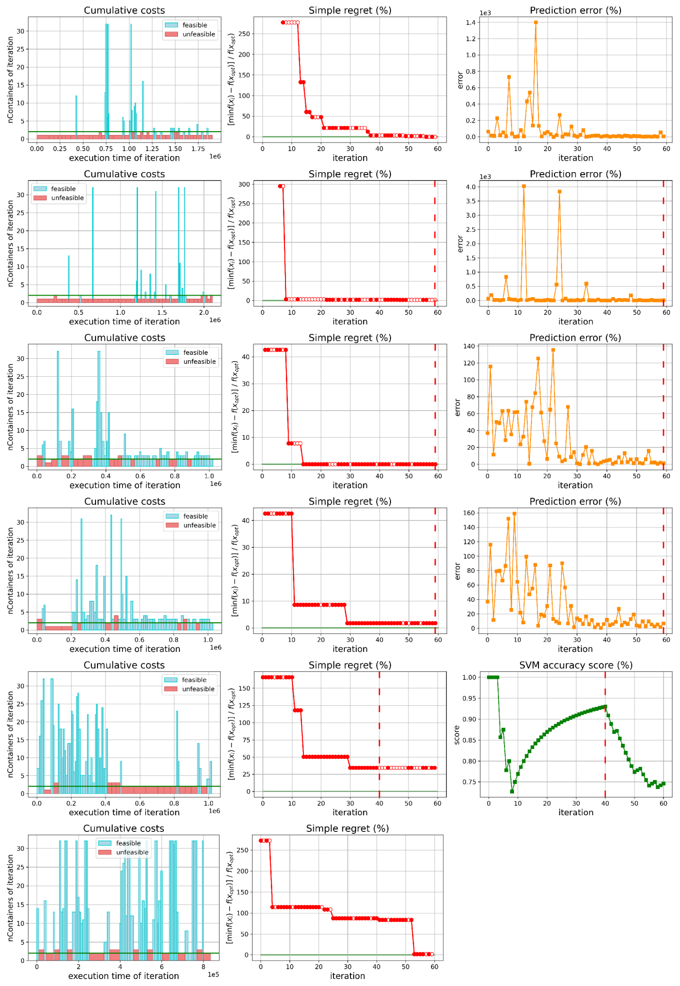


Fig. 6. Stereomatch: comparison of CherryPick pure BO (variant A), MALIBOO variants B, C, and D, SVM-CBO, and OpenTuner, respectively, from top to bottom.

TABLE VI
MEASURED METRICS FOR GPU EXPERIMENTS

scenario	var. A	var. B	OT
GPU benchmarks			
BFS mean feasible cost	1	0.941	1.003
MD mean feasible cost	1	0.970	0.996

still be used in an unconstrained minimization scenario if the baseline is regular EI, as explained in Section VI-A. We cannot use variants C or D, as there is no constraint to compute the indicator function. For the same reason, we cannot compare against SVM-CBO, as the algorithm only performs constrained optimization. Finally, we do not employ any termination criterion in this experiment.

We show the average cost of feasible configurations in Table VI. Fig. 7 displays the results for an example MALIBOO run for both GPU applications. Since we are not minimizing core- or VM-related costs, nor is there any constraint in this particular scenario, we do not show the bar plot for cumulative costs. In the BFS application (Fig. 7(a)), the simple regret of the proposed algorithm is competitive compared to variant A (pure BO) and OpenTuner at around 10%, whereas in MD (Fig. 7(b)) it converges to 5% much faster than pure BO, similarly to OpenTuner. Moreover, the absolute error of the ML models stays within 15% for over 80% of all algorithm iterations. The results of these applications show that our algorithm also successfully applies to unconstrained optimization scenarios.

D. Molecular Docking Application

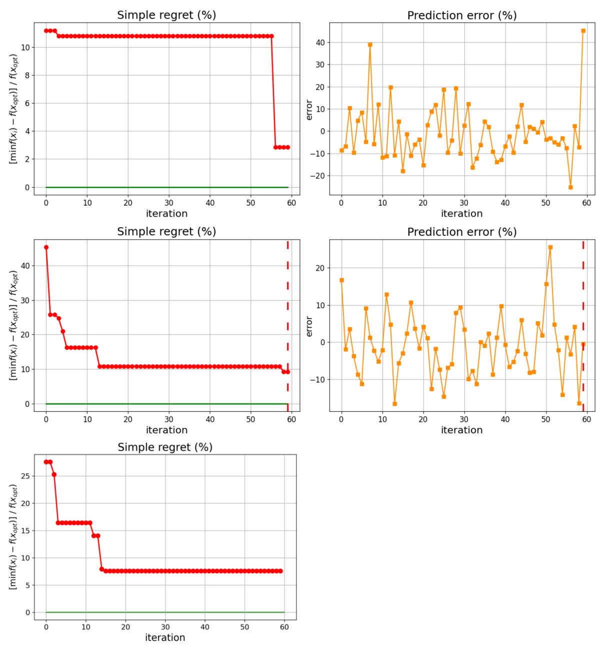
With the LiGen application, we tackle a more complicated optimization problem, for which the assumptions of the CherryPick approach do not hold. In this case, we use a complex objective function to minimize multiple relevant quantities simultaneously. Given the configuration vector x, suppose that $R(x)$ is the average RMSD metric of the resulting solution (as described in Section VI-B4), and $T(x)$ is the execution time of the application. We can formulate this optimization problem as follows:

$$\min_{x\in\mathcal{A}} R^3(x)\,T(x)+\varepsilon$$
$$\text{s.t. } R(x)\le R_{\max} \tag{12}$$

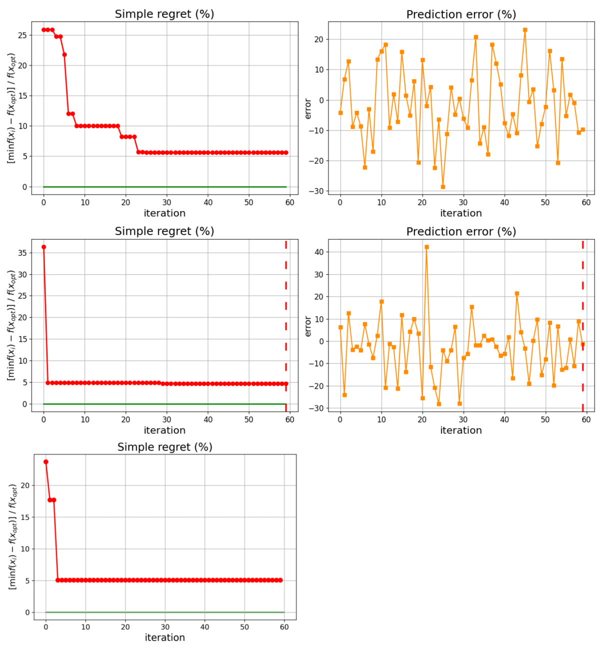
In this case, the constraint function is $g(\cdot)\equiv R(\cdot)$, which we want to keep under a certain threshold $R_{\max}$ to guarantee a minimum degree of quality of the solution found. However, the non-linearity of the target function $f(x)=R^3(x)\,T(x)$ in equation (12) forbids the use of the CherryPick model, which requires a linear relationship between $f(\cdot)$ and $R(\cdot)$, and therefore a Gaussian probability distribution on the values of the latter. In some cases, authors assume another Gaussian Process for the constraint function, which is independent of the one for the objective function [8], [27], [28]. However, this approach is not viable in this case because of the explicit dependence of $f(\cdot)$ on $R(\cdot)$.

Therefore, we compare pure BO with the Expected Improvement (EI) acquisition function described in equation (6) with an ML-integrated acquisition function implemented in MALIBOO. In particular, we model the RMSD $R(\cdot)$ with an ML regressor $\widehat{R}(\cdot)$, and we multiply regular EI by the indicator function $I\{\widehat{R}(x)\le R_{\max}\}$. This is the $g_C(\cdot)$ acquisition function described in Section VI-A1 when $g_A(\cdot)$ is the regular EI function. As always, we initialize all algorithms with the same points. We run this comparison with 10 representative $R_{\max}$ values ranging from 2.00 to 2.90 (which are relevant values for the LiGen application domain) and 10 different sets of initial points. We then average the results of the corresponding 100 tests. Comparison with SVM-CBO and OpenTuner is possible since they do not hold any particular assumption on the optimized function or the constraints. As mentioned, we do not use termination criteria for MALIBOO given the large size of the optimization domain, which requires extensive exploration.

Fig. 8 shows a representative example of a graphical comparison. We also collect the average metrics for the experiments in Table VII. The overall results of our LiGen experiments indicate that MALIBOO reduces the average number of unfeasible configurations by almost half. This is a consequence of properly including information from the constraint into the



(a) BFS application.



(b) MD application.

Fig. 7. Unconstrained optimization: comparison of pure BO (variant A, top row), MALIBOO variant B (middle row) and OpenTuner (bottom row) in the BFS (7a) and MD (7b) GPU applications.

TABLE VII
MEASURED METRICS FOR LIGEN EXPERIMENTS

scenario	var. A	var. C	SVM	OT
LiGen				
unfeasible runs	28.6	16.9	26.0	26.0
mean feasible regret	0.38	0.34	2.07	1.47

acquisition function. The average regret of feasible executions also decreases compared to pure BO. These statistics show the benefits of a proper ML-integrated approach, which is usable even when we do not fulfill the assumptions of the CherryPick model. Indeed, the ML models are reasonably accurate in predicting the RMSD of chosen configurations, as evidenced by the prediction MAPE rarely exceeding 15%. On the other hand, the SVM-CBO algorithm often struggles to get closer to the optimum despite a decent performance of its SVM model and is stuck to an average regret of about 200%. Similar considerations hold for OpenTuner, with an average regret of about 150%.

Note that the *optimal configuration* used in this analysis when evaluating the regret refers to the best configuration we found by testing 3142 configurations running the LiGen application for three weeks. Indeed, the LiGen parameter space is too large for the *true* best configuration to be known. With each application execution taking about 5 to 10 minutes, the full exploration of the configuration space would need at least 580 years' worth of computing time.

VIII. DISCUSSION

The experimental evidence of the previous sections shows the robustness of the proposed approach, especially in the LiGen case, in which we have complex software, a challenging optimization problem, and a vast search space. The predicting capabilities of ML can help in guiding the exploration process of BO in several ways, for instance, by encouraging points with promising values of the objective function (as in $\widetilde{a}_1$ / variant B), or by excluding points which are projected to be unfeasible (as in $\widetilde{a}_2$ / variant C). In particular, variant B is suitable for unconstrained and constrained optimization scenarios, while other variants do not apply to the unconstrained case.

In cases where we can use multiple acquisition functions, they have varying degrees of performance, although they all outperform the benchmark techniques from the literature. Specifically, according to the figures of merit reported in Table II, for the Query26 and Kmeans Spark applications, variant B achieves overall better results, while variant C stands out in the SparkDL case. For the Stereomatch edge computing application, variants C and D perform significantly better than B (see Table V). This fact is consistent with a well-known empirical observation in the ML community: the performance of any data-driven approach is often dependent on the specific task at hand, with different models faring better than others when used in different scenarios.

The ML-integrated acquisition functions also outperform pure BO in the other scenarios. In particular, variant B is superior in terms of simple regret in the unconstrained optimization of GPU benchmarks. In contrast, variant C is superior in the LiGen case, where we have a more complex optimization function and a massively large search space.

An overall comparison with the other literature methods SVM-CBO and OpenTuner reveals that the VM flavor analysis is the only case in which MALIBOO variants do not stand out, with SVM-CBO having comparable performance and OpenTuner

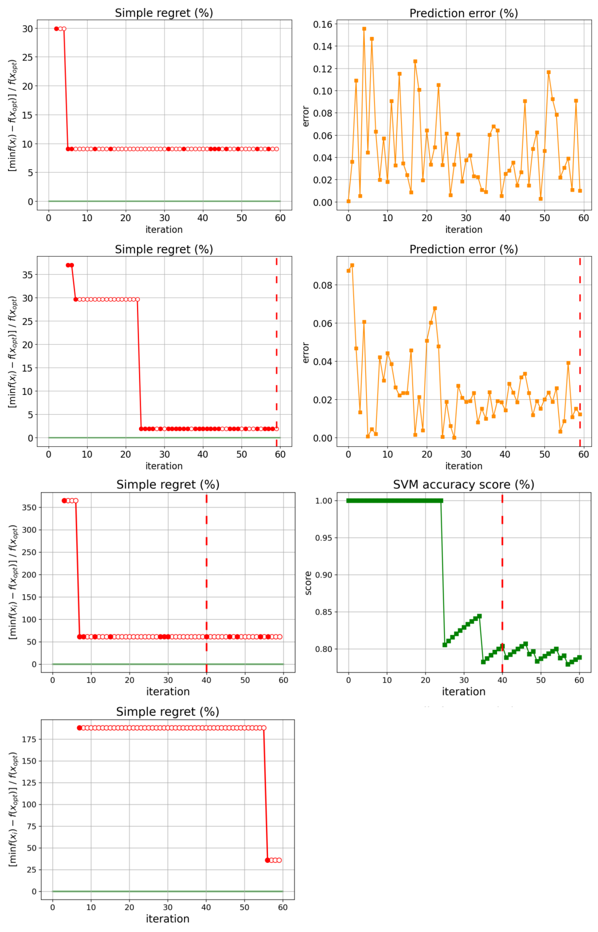


Fig. 8. LiGen: comparison of pure BO, MALIBOO, SVM-CBO, and OpenTuner, respectively, from top to bottom.

achieving slightly better results. In all other cases, SVM-CBO and OpenTuner are considerably less effective regarding unfeasible executions and costs, which are larger by up to a factor of 3–4 compared to MALIBOO.

IX. Conclusions and Future Work

In this work, we presented MALIBOO, an approach combining the BO algorithm with ML techniques, to find an optimal configuration of recurring jobs running in public or private clouds for a wide range of applications when subject to resource constraints. Results on the tested applications, including scientific computing and Big Data software, show that our algorithms significantly reduce the number of unfeasible executions compared to competing approaches, including the pure BO-based CherryPick, the ML-based SVM-CBO, and the ensemble-based OpenTuner, and decrease the average cost of the chosen configuration. Overall, each of our algorithm variants outperforms the state-of-the-art techniques used for comparison.

Future works are towards exploring algorithm hyperparameters and acquisition functions when dealing with large computing environments, thus enabling parallel exploration of alternative configurations.

References

[1] L. Wang et al., “Morphling: Fast, near-optimal auto-configuration for cloud-native model serving,” in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 639–653.

[2] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies, “Arrow: Low-level augmented Bayesian optimization for finding the best cloud VM,” in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 660–670.

[3] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, “CherryPick: Adaptively unearthing the best cloud configurations for Big Data analytics,” in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 469–482.

[4] E. Barbierato, M. Gribaudo, and M. Iacono, “Modeling apache hive based applications in Big Data architectures,” in *Proc. 7th Int. Conf. Perform. Eval. Methodol. Tools*, 2013, pp. 30–38.

[5] A. Gandini, M. Gribaudo, W. J. Knottenbelt, R. Osman, and P. Piazzolla, “Performance evaluation of NoSQL databases,” in *Proc. 11th Eur. Workshop Comput. Perform. Eng.*, Springer, 2014, pp. 16–29.

[6] E. Gianniti, M. Ciavotta, and D. Ardagna, “Optimizing quality-aware Big Data applications in the cloud,” *IEEE Trans. Cloud Comput.*, vol. 9, no. 2, pp. 737–752, Second Quarter, 2021.

[7] P. I. Frazier, “A Tutorial on Bayesian Optimization,” 2018, *arXiv: 1807.02811*.

[8] T. Pourmohamad and H. K. Lee, “Bayesian optimization via barrier functions,” *J. Comput. Graphical Statist.*, vol. 31, no. 1, pp. 74–83, 2022.

[9] M. Schonlau, W. J. Welch, and D. R. Jones, “Global versus local search in constrained optimization of computer models,” *IMS Lecture Notes-Monograph Ser.*, vol. 34, pp. 11–25, 1998.

[10] M. Lattuada, E. Gianniti, D. Ardagna, and L. Zhang, “Performance prediction of deep learning applications training in GPU as a service systems,” *Cluster Comput.*, vol. 25, no. 2, pp. 1279–1302, 2022.

[11] A. Maros et al., “Machine learning for performance prediction of spark cloud applications,” in *Proc. IEEE 12th Int. Conf. Cloud Comput.*, 2019, pp. 99–106.

[12] P. Nawrocki and P. Osypanka, “Cloud resource demand prediction using machine learning in the context of QoS parameters,” *J. Grid Comput.*, vol. 19, no. 2, pp. 1–20, 2021.

[13] D. F. Kirchoff, M. Xavier, J. Mastella, and C. A. De Rose, “A preliminary study of machine learning workload prediction techniques for cloud applications,” in *Proc. 27th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process.*, 2019, pp. 222–227.

[14] B. Guindani, D. Ardagna, and A. Guglielmi, “MALIBOO: When machine learning meets Bayesian Optimization,” in *Proc. IEEE 7th Int. Conf. Smart Cloud*, 2022, pp. 1–9.

[15] J. Ansel et al., “OpenTuner: An extensible framework for program autotuning,” in *Proc. 23rd Int. Conf. Parallel Archit. Compilation Techn.*, 2014, pp. 303–316.

[16] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *Proc. 25th ACM Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 2623–2631.

[17] T. S. Frisby, Z. Gong, and C. J. Langmead, “Asynchronous parallel Bayesian optimization for AI-driven cloud laboratories,” *Bioinf.*, vol. 37, no. Supplement 1, pp. i451–i459, 2021.

[18] Q. Li et al., “RAMBO: Resource allocation for microservices using Bayesian optimization,” *IEEE Comput. Archit. Lett.*, vol. 20, no. 1, pp. 46–49, Jan./Jun. 2021.

[19] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, “Google vizier: A service for black-box optimization,” in *Proc. 23rd ACM Int. Conf. Knowl. Discov. Data Mining*, 2017, pp. 1487–1495.

[20] S. Cereda, S. Valladares, P. Cremonesi, and S. Doni, “GPTuner: A contextual Gaussian process bandit approach for the automatic tuning of IT Configurations under varying workload conditions,” in *Proc. VLDB Endowment*, vol. 14, no. 8, pp. 1401–1413, 2021.

[21] J. Bergstra, D. Yamins, and D. Cox, “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures,” in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2013, pp. 115–123.

[22] F.-J. Willemsen, R. Van Nieuwpoort, and B. Van Werkhoven, “Bayesian optimization for auto-tuning GPU kernels,” in *Proc. Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst. Supercomput.*, 2021, pp. 106–117.

[23] M. Lindauer et al., “SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization,” *J. Mach. Learn. Res.*, vol. 23, no. 54, pp. 1–9, 2022.

[24] M. Bilal, M. Serafini, M. Canini, and R. Rodrigues, "Do the best cloud configurations grow on trees? An experimental evaluation of black box algorithms for optimizing cloud workloads," in *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2563–2575, 2020.

[25] L. Fang, E. Makkonen, M. Todorović, P. Rinke, and X. Chen, "Efficient amino acid conformer search with Bayesian optimization," *J. Chem. Theory Comput.*, vol. 17, no. 3, pp. 1955–1966, 2021.

[26] S. Jalas et al., "Bayesian optimization of a laser-plasma accelerator," *Phys. Rev. Lett.*, vol. 126, no. 10, 2021, Art. no. 104801.

[27] S. Ariafar, J. Coll-Font, D. H. Brooks, and J. G. Dy, "ADMMBO: Bayesian optimization with unknown constraints using ADMM," *J. Mach. Learn. Res.*, vol. 20, no. 123, pp. 1–26, 2019.

[28] M. A. Gelbart, J. Snoek, and R. P. Adams, "Bayesian optimization with unknown constraints," in *Proc. 30th Conf. Uncertainty Artif. Intell.*, 2014, pp. 250–259.

[29] X. Pan, S. Venkataraman, Z. Tai, and J. Gonzalez, "Hemingway: Modeling distributed optimization algorithms," in *Proc. Workshop ML Syst. NeurIPS*, 2016. [Online]. Available: https://arxiv.org/abs/1702.05865 for the reference and https://web.archive.org/web/20170329170507/https://sites.google.com/site/mlsysnips2016/home for the event

[30] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in *Proc. IEEE 13th Int. Conf. Cloud Comput.*, 2020, pp. 609–618.

[31] Y. Ding, A. Pervaiz, S. Krishnan, and H. Hoffmann, "Bayesian learning for hardware and software configuration co-optimization," Univ. Chicago Dept. Comput. Sci., Tech. Rep. TR-2020–13, 2020.

[32] R. Perego, A. Candelieri, F. Archetti, and D. Pau, "Tuning deep neural network's hyperparameters constrained to deployability on tiny systems," in *Proc. Int. Conf. Artif. Neural Netw.*, Springer, 2020, pp. 92–103.

[33] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwaenepoel, and D. Garlan, "Lynceus: Cost-efficient tuning and provisioning of data analytic jobs," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst.*, 2020, pp. 56–66.

[34] C. K. Williams and C. E. Rasmussen, *Gaussian Processes for Machine Learning*, vol. 2. Cambridge, MA, USA: MIT Press, 2006.

[35] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," *Adv. Neural Inf. Process. Syst.*, vol. 25, pp. 2951–2959, 2012.

[36] P. Hennig and C. J. Schuler, "Entropy search for information-efficient global optimization," *J. Mach. Learn. Res.*, vol. 13, no. 6, pp. 1809–1837, 2012.

[37] X. Luo, "Minima distribution for global optimization," 2018, *arXiv: 1812.03457*.

[38] B. Guindani, M. Lattuada, and D. Ardagna, "AMLLibrary: An AutoML approach for performance prediction," in *Proc. 37th Int. Conf. Modelling Simul.*, vol. 37, pp. 241–247, 2023.

[39] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Comput. J.*, vol. 7, no. 4, pp. 308–313, 1965.

[40] D. Cvijović and J. Klinowski, "Taboo search: An approach to the multiple-minima problem for continuous functions," in *Handbook of Global Optimization*, Berlin, Germany: Springer, 2002, pp. 387–406.

[41] M. Zaharia et al., "Apache spark: A unified engine for Big Data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[42] E. Paone et al., "An exploration methodology for a customizable OpenCL stereo-matching application targeted to an industrial multi-cluster architecture," in *Proc. IEEE/ACM/IFIP 8th Int. Conf. Hardware/Softw. Codesign Syst. Synth.*, 2012, pp. 503–512.

[43] D. Gadioli et al., "EXSCALATE: An extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight SARS-CoV-2," *IEEE Trans. Emerg. Topics Comput.*, vol. 11, no. 1, pp. 170–181, First Quarter, 2023.

[44] G. Chen, H. Meng, Y. Liang, and K. Huang, "GPU-accelerated real-time stereo estimation with binary neural network," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2896–2907, Dec. 2020.

[45] F. Filippini, M. Lattuada, M. Ciavotta, A. Jahani, D. Ardagna, and E. Amaldi, "A path relinking method for the joint online scheduling and capacity allocation of DL training workloads in GPU as a service systems," *IEEE Trans. Services Comput.*, vol. 16, no. 3, pp. 1630–1646, May/Jun. 2023.

[46] G. H. Golub and J. M. Ortega, *Scientific Computing: An Introduction With Parallel Computing*. Amsterdam, The Netherlands: Elsevier, 2014.

[47] E. Vitali, D. Gadioli, G. Palermo, A. Beccari, C. Cavazzoni, and C. Silvano, "Exploiting OpenMP and OpenACC to accelerate a geometric approach to molecular docking in heterogeneous HPC nodes," *J. Supercomput.*, vol. 75, no. 7, pp. 3374–3396, 2019.

[48] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

Bruno Guindani (Graduate Student Member, IEEE) received the MSc degree in mathematical engineering from the Politecnico di Milano, Italy, in 2021. He is currently working toward the PhD degree in computer science with the Department of Electronics, Information, and Bioengineering, Politecnico di Milano, Italy. He currently works in the context of the LIGATE EuroHPC project. His research interests include the application of Bayesian statistical models, e.g., Bayesian optimization, and machine learning techniques for the optimization of HPC systems.

Danilo Ardagna (Senior Member, IEEE) received the PhD degree in computer engineering from Politecnico di Milano, in 2004. He is associate professor with the Department of Electronics, Information, and Bioengineering, Politecnico di Milano, Italy. His work focuses on performance modeling and the design, prototype, and evaluation of optimization algorithms for resource management of cloud computing systems.

Alessandra Guglielmi received the PhD degree in mathematics from the University of Milano, Italy, in 1997. She is full professor in statistics with Politecnico di Milano. Her recent research interests are Bayesian mixture models for density estimation and clustering and Bayesian Optimization. From the application point of view, her work concerns statistics for medicine, environmental applications, Bayesian optimization for cloud systems, and Big Data applications.

Roberto Rocco (Graduate Student Member, IEEE) received the MSc degree in computer science engineering from Politecnico di Milano, Italy in 2020. He is currently working toward the PhD degree with the Department of Electronics, Information, and Bioengineering, Politecnico di Milano, Italy. His research interests include fault tolerance and software adaptivity in HPC, focusing on the autotuning scenario.

Gianluca Palermo (Senior Member, IEEE) received the MSc degree in electronic engineering, in 2002, and the PhD degree in computer engineering, in 2006, from Politecnico di Milano, Italy. He is currently full professor with the Department of Electronics, Information, and Bioengineering, Politecnico di Milano, Italy. His research interests include design methodologies and architectures for embedded and HPC systems, focusing on autotuning and high-throughput molecular docking.

Open Access funding provided by 'Politecnico di Milano' within the CRUI CARE Agreement