An Artificial Bee Colony algorithm with Machine Learning for Constrained Optimization in HPC

Roberto Sala*, Nikita Litovchenko†, Davide Gadioli*, Gianluca Palermo* and Danilo Ardagna*
Department of Electronics, Information and Bioengineering, Politecnico di Milano, Milan, Italy
*name.surname@polimi.it †nikita.litovchenko@mail.polimi.it

***Abstract*—High-Performance Computing (HPC) is a fundamental tool for tackling complex scientific and engineering problems. Optimizing applications for the heterogeneous and massively parallel nature of modern HPC hardware is essential for achieving timely and resource-efficient results. This paper introduces ABC-MLCO, a novel Artificial Bee Colony (ABC) algorithm enhanced with machine learning for constrained optimization in discrete configuration spaces, specifically targeting HPC scenarios with black-box performance metrics. ABC-MLCO extends the original ABC algorithm with mechanisms that define the feasible region, promote escape from local optima, prevent redundant evaluations, and intensify exploration and exploitation. We first evaluate the algorithm on benchmark functions, observing improvements in regret, feasibility rate, and convergence speed over the original ABC. Then, targeting a virtual screening application for drug discovery, ABC-MLCO outperforms well-known state-of-the-art methods in terms of final regret, achieving average improvements up to 93%.**

***Index Terms*—Artificial Bee Colony, Optimization, Discrete Variables, Black-box Constraints, Machine Learning, HPC**

I. Introduction

High-Performance Computing (HPC) has become an essential tool for addressing increasingly complex scientific and engineering challenges, ranging from climate modeling and drug discovery to materials science and astrophysics. The continuous evolution of HPC hardware, characterized by heterogeneous architectures and massive parallelism, presents unprecedented opportunities and significant challenges for efficient application execution. While computational power continues to grow, mapping complex workloads onto such intricate hardware systems remains complex. As a result, optimizing HPC applications is crucial to ensure timely and resource-efficient scientific discovery and technological advancement. For example, two of the largest virtual screening campaigns have been conducted to evaluate billions of potential drug candidates against multiple SARS-Cov-2 targets [1], [2].

One of the primary goals of HPC is to maximize application throughput, measured in terms of floating-point operations per second (FLOPS). This objective becomes even more critical in scenarios where solutions are needed within strict time constraints, making HPC capabilities essential for rapid response and decision-making. Despite its importance, current approaches to jointly optimize the configuration of the HPC system and the target application face several limitations. Manual tuning is time-consuming and requires deep domain expertise, while architecture-specific optimizations often sacrifice portability across different platforms. Furthermore, the increasing complexity of applications and hardware architectures makes it more challenging to exploit all available performance effectively. Additionally, many optimization algorithms still struggle to utilize the extensive parallelism of modern HPC systems, which restricts scalability and speedup even as core counts and accelerator usage increase.

Swarm Intelligence algorithms are inspired by the decentralized, collective behaviors observed in natural swarms. These algorithms are inherently parallel [3]: each agent computes independently and concurrently, without a central coordinator. This decentralized interaction not only promotes scalability and fault tolerance but also efficiently integrates with modern multi-core, GPU, and distributed HPC platforms [3]. In particular, we focus on the Artificial Bee Colony (ABC) algorithm [4], which is inspired by the foraging behavior of honeybees. By assigning agents to different roles—such as employed bees, onlookers, and scouts—the ABC algorithm provides significant flexibility in balancing exploration and exploitation. However, ABC was originally devised for unconstrained, continuous optimization and can suffer from premature convergence to local optima.

This paper introduces ABC-MLCO, an Artificial Bee Colony (ABC) algorithm enhanced with Machine Learning (ML) techniques for constrained optimization in discrete configuration spaces. ABC-MLCO is designed to handle black-box constraints—such as execution-time limits and minimum accuracy requirements—making it well suited for HPC scenarios. To adapt the original ABC algorithm to this scenario, we introduce several key enhancements: (i) ML models are used to approximate the feasible region and guide the search toward optimal solutions; (ii) A local memory mechanism not only avoids redundant evaluations of the Objective Function (OF) but also provides training data for the ML models; (iii) we adopt the Lévy flight to enhances exploration and helps prevent premature convergence to local optima.

To validate our approach, we assess the proposed algorithm on benchmark functions, demonstrating improvements in final regret, feasibility rate, and convergence speed over the original ABC algorithm. We then conduct an autotuning campaign on *LiGen*, the molecular docking application from the EXSCALATE virtual screening platform [2], to assess the proposed approach in a real-world case study. *LiGen* offers a wide range of parameters that impact the trade-off between quality and throughput, and optimizes its execution based on the underlying hardware. We compare our algorithm with state-

of-the-art approaches, including EMaliboo [5] and OpenTuner [6]. On average, ABC-MLCO reduces regret, respectively, by 52% and 93%. This comes at the cost of a 10% lower feasibility rate than the former but 46% higher than the latter.

The remainder of this work is organized as follows. Section II reviews related work. Section III outlines the problem under study. Section IV introduces the original ABC framework. Section V details our ABC-MLCO algorithm. Section VI reports the experimental results. Finally, Section VII presents conclusions and directions for future work.

II. Related Work

The ABC algorithm has garnered significant attention in recent years. In a comprehensive comparative study, [4] demonstrated its competitive performance against other well-established evolutionary and swarm intelligence algorithms such as Genetic Algorithms, Particle Swarm Optimization, Differential Evolution, and Evolution Strategies. This algorithm is inspired by the foraging behavior of honeybees, a topic explored in several foundational studies. For example, [7] developed a model using reaction-diffusion equations to simulate how communication within a honeybee colony influences its foraging dynamics, effectively capturing the key elements of food source recruitment and abandonment. Further research by [8] emphasized the importance of information exchange among foragers, showing that enhanced communication improves collective decision-making and facilitates the selection of the most profitable food sources.

Subsequent research has aimed at enhancing the performance of the ABC algorithm. [9] proposed an Information Learning-based ABC, which incorporates subpopulation division, dynamic population size adjustment, and specialized search mechanisms to improve information exchange. In tackling the issue of slow convergence in the original ABC algorithm, [10] introduced the Directed ABC approach, which integrates directional information—a parameter that tracks the last direction of improvement for each dimension—to guide the search process. Additionally, [11] developed NSABC, an improved ABC algorithm which employs a neighborhood selection mechanism to address the limitations of probability-based selection during the onlooker bee phase and utilizes opposition-based learning to enhance the scout bee phase.

The ABC algorithm has been applied in various domains, particularly in HPC and edge-cloud computing. For instance, [12] utilized the ABC algorithm for intelligent computation offloading in IoT applications within scalable edge computing environments. [13] proposed an ABC-based fog computing resource scheduling strategy aimed at minimizing time delays and energy consumption in IoT systems. Lastly, [14] presented TRACTOR, a multi-objective virtual machine placement scheme for edge-cloud data centers that employs the ABC algorithm to optimize both power consumption and network traffic.

III. Problem Overview

Consider the following optimization problem:

$$\begin{aligned} &\min_{\boldsymbol{x}\in D} f(\boldsymbol{x}) + \eta \\ &\text{s.t. } g_c(\boldsymbol{x}) \in [G^c_{\min}, G^c_{\max}] \; \forall c = 1, \ldots, C. \end{aligned} \tag{1}$$

In this problem, we seek to minimize a black-box OF, $f : D \to \mathbb{R}$, over the discrete domain $D \subset \mathbb{R}^d$, where the analytical form of f is unknown and properties such as concavity, linearity, or differentiability cannot be assumed. The function f might measure the performance of an Artificial Intelligence (AI) model during training or the cost of operating a service in an Edge–Cloud continuum. Here, d is the number of input features. Additionaly, we include a noise term η to capture variability in measuring the OF. The feasible domain is defined as $D' = \{\boldsymbol{x} \in D \mid g_c(\boldsymbol{x}) \in [G^c_{\min}, G^c_{\max}] \; \forall c = 1, \ldots, C\}$, where C is the number of black-box constraints. For brevity, we define $\boldsymbol{g}(\cdot)$ as the vector of constraint functions (CFs) $g_c(\cdot)$. In this context, the CFs might encode limits such as the maximum inference or execution time of the model or application, or any other Quality-of-Service (QoS) requirement. Assume that each $g_c(\cdot)$ is also black box. This imply that information about $f(\cdot)$ and $g(\cdot)$ can only be obtained through direct evaluation.

In this context, we assume the availability of N servers operating concurrently to optimize the OF. Given the inherent parallelism of the ABC algorithm and its minimal reliance on central coordination, it is well-suited for this problem. Furthermore, we require an algorithm that can converge quickly and has low update times. The ABC algorithm meets all of these requirements.

IV. Artificial Bee Colony Algorithm Background

The ABC algorithm was introduced in [4], inspired by the foraging behaviour model of a honeybee colony [7], [8]. In this section, we provide an overview of the behaviour of real bees in a hive and a summary of the original ABC algorithm.

A. Foraging Behavior in Honeybee Colonies

The foraging strategy of a honeybee colony is a striking example of decentralized collective intelligence, allowing efficient exploitation of scattered nectar sources. Mathematical models aim to capture the essential dynamics of this system [7], [8]. These models typically define three core components—*food sources*, *employed foragers*, and *unemployed foragers*—and two primary behavioral modes: *recruitment* and *abandonment* of food sources.

Food Sources: The profitability of a food source is evaluated by bees based on multiple factors, including its distance from the hive, nectar richness, and ease of extraction. For modeling simplicity, this complex assessment is often condensed into a single profitability value. Research suggests this value corresponds to the net energetic efficiency—the energy gained minus the energy cost of foraging—effectively balancing richness and distance. The colony seeks to dynamically identify and exploit optimal resources.

Employed Foragers: These bees are committed to exploiting a specific food source. Upon returning to the hive, an

employed forager carries information about the location of the source (distance and direction) and its profitability. She conveys this information primarily through the *waggle dance*, which takes place on the *dance floor* of the hive. The duration and vigor (i.e., number of waggle runs) of her dance correlate strongly with source profitability—more profitable sources lead to longer, more vigorous, and more frequent dances. Notably, employed foragers have local knowledge, related to the source they are currently exploiting.

Unemployed Foragers: These bees actively search for a food source to exploit and exist in two primary states:

- **Scouts**: These foragers explore the environment randomly, potentially discovering previously unknown rich sources. Scouts make up about 5-10% of the population.
- **Onlookers**: These bees remain within the hive, typically in proximity of the dance floor, waiting to receive information about currently exploited sources from the waggle dances of returning employed foragers.

Recruitment and Information Exchange: Once an employed forager returns to the hive, recruitment and information exchange take place primarily on the dance floor. Communication via the waggle dance is central to recruitment. Onlookers often do not watch entire dances (which encode profitability in their duration) but instead tend to follow a single dancer, potentially chosen randomly, before departing for the indicated source. The colony achieves efficiency because foragers from highly profitable sources dance more intensely and more frequently. As a result, at any given moment, the majority of dancing bees are advertising the most profitable sources, increasing the likelihood that an onlooker follows a high-quality lead. Thus, the overall recruitment rate to a food source is effectively proportional to its profitability.

Abandonment: The counterpart to recruitment is the abandonment of a food source. An employed forager ceases visiting her current source, typically when it becomes depleted or significantly less profitable, and transitions back into the unemployed state. Biologically realistic interpretations suggest that abandonment probability is inversely proportional to the profitability of the source.

Through the continuous interplay of scouting, foraging, dancing, and abandonment, the colony dynamically allocates its workforce to the most rewarding sources.

B. The ABC algorithm

In the ABC framework, potential solutions to an optimization problem are represented as *food source* positions in a d-dimensional search space, with the quality or *fitness* of a solution corresponding to the *nectar amount* of its associated food source. Let $\boldsymbol{x}_i^{(t)}$ denote the position of bee i at iteration t. The algorithm maintains a population of N bees that alternate between employed and onlooker phases. Recall that the original ABC algorithm was designed to address unconstrained problems, where $\boldsymbol{x} \in \mathcal{D}$.

After initializing a population of random solutions $\boldsymbol{x}_i^{(0)}\ \forall i$ within predefined search-space boundaries, the algorithm proceeds iteratively through three main phases:

1) **Employed Bee Phase**: Each employed bee is associated with a specific solution (food source) and performs a local search by generating a candidate solution $\hat{\boldsymbol{x}}_i^{(t+1)}$ in the neighborhood of its current position. This search typically involves modifying the current best solution $\boldsymbol{x}_i^{(t)}$ along a randomly chosen direction j, based on the difference between it and another randomly selected solution $k \neq i$ from the population, as follows:
$$\hat{x}_{ij}^{(t+1)} = x_{ij}^{(t)} + \phi_{ij}^{(t+1)}(x_{ij}^{(t)} - x_{kj}^{(t)}), \tag{2}$$
where $x_{ij}^{(t)}$ denotes the best position of bee i along direction j at iteration t, and $\phi_{ij}^{(t+1)} \sim \mathcal{U}(-1, 1)$. A greedy selection mechanism is then applied: if the new candidate solution is better than or equal to the current best one, the employed bee memorizes the new position; otherwise, it retains the old solution.
2) **Onlooker Bee Phase**: After the employed bees complete their searches, onlooker bees probabilistically select which employed bee's food source to explore further. Selection is typically performed using a roulette wheel mechanism, where solutions with higher fitness have a greater probability of being chosen. The fitness function is computed as follows:
$$fit_i^{(t)} = \begin{cases} \frac{1}{1+f(\boldsymbol{x}_i^{(t)})}, \text{ if } f(\boldsymbol{x}_i^{(t)}) \geq 0 \\ 1 + |f(\boldsymbol{x}_i^{(t)})|, \text{ otherwise} \end{cases} \tag{3}$$
while the selection probability distribution $s^{(t)}$ for each employed bee i is computed as follows:
$$s^{(t)} = \frac{fit_i^{(t)}}{\sum_{i=1}^{N} fit_i^{(t)}}. \tag{4}$$
Once an onlooker selects a source, it performs a similar local search and greedy selection process as in (2), potentially improving the chosen solution.
3) **Scout Bee Phase**: To maintain diversity and prevent premature convergence, the ABC algorithm includes an exploration mechanism. If a solution associated with an employed bee fails to improve after a predefined number of trials Λ, it is considered *abandoned*. The corresponding employed bee then becomes a scout and generates a completely new random solution within the search space boundaries, replacing the abandoned one:
$$\boldsymbol{x}_i^{(t+1)} \sim \mathcal{U}(D). \tag{5}$$

This cycle of employed, onlooker, and scout phases repeats until a termination criterion is met, such as a maximum number of iterations. By balancing exploitation (refining good solutions) and exploration (introducing diversity), the ABC algorithm efficiently searches the solution space for optimal or near-optimal results.

V. ABC-MLCO

In this section, we present the modifications proposed to the original ABC algorithm to address (1). Specifically, we

introduce adaptations to handle black-box constraints (Section V-A), enhance the exploitation capability of the ABC algorithm using ML (Section V-B), improve exploration to escape local optima through Lévy flights (Section V-C), and reduce repetitions in the discrete domain (Section V-D). Finally, we present the ABC-MLCO algorithm (Section V-E).

A common feature across all proposed modifications is the use of memory: each bee is equipped with a record of its last H visited points, including their positions, OF values, and CF values, similar to the approach in [15]. In the following sections, we describe how this memory is leveraged to improve the efficiency of the ABC algorithm.

A. Dealing with Black-box Constraints

In an HPC setting, black-box constraints often represent QoS requirements that must be satisfied, such as the inference time of an AI model or the minimum precision of a prediction model on a validation set. The resulting CFs typically cannot be expressed analytically, and no information is available regarding their concavity, linearity, or derivatives. For this reason, information about the CFs $g(\cdot)$ can only be obtained through direct measurement. To handle black-box constraints, we propose two modifications to the original ABC algorithm. First, we modify the fitness function (3) so that the fitness of an infeasible observed point is set to zero:

$$\tilde{fit}_i^{(t)} = \begin{cases} fit_i^{(t)}, \text{ if } \boldsymbol{x}_i^{(t)} \in D' \\ 0, \text{ otherwise.} \end{cases} \tag{6}$$

This implies that the onlooker bees avoid exploiting regions that are likely unfeasible. The second modification involves using the *local* memory of each bee to explore the feasible region. In particular, during the scout phase, a bee selects the next point to explore based on its previous evaluations, instead of purely randomly as in (5). This selection is guided by an ML model (either Regression, Random Forest or Neural Networks) trained on the H most recent observations made by the bees:

$$\boldsymbol{x}_i^{(t+1)} \sim \mathcal{U}(\tilde{D}') \tag{7}$$

where $\tilde{D}' = \{\boldsymbol{x} \in D \mid \tilde{g}_{ic}(\boldsymbol{x}) \in [G^c_{\min}, G^c_{\max}] \; \forall c = 1, \ldots, C\}$, and $\tilde{\boldsymbol{g}}_i(\cdot)$ is a surrogate ML model used to estimate the CFs $\boldsymbol{g}(\cdot)$. The model is trained on the *local* memory set $\left\{\left(\boldsymbol{x}_i^{(h)}, \boldsymbol{g}(\boldsymbol{x}_i^{(h)})\right), \; h = 1, \ldots, H\right\}$ of bee i, similarly to the approach we proposed in [16].

B. ML models to enhance exploitation

Memory is also a valuable resource for navigating the domain and identifying the optimal solution to (1). For this reason, we propose modifying the selection process for the next point to evaluate (2), leveraging the predictive capability of ML models. Specifically, while the direction j and the other bee k are still chosen randomly, the coefficient $\phi_{ij}^{(t+1)}$ is selected to maximize the value of $\tilde{f}_i(\cdot)$, a surrogate ML model approximating the OF, trained on the local memory set $\left\{\left(\boldsymbol{x}_i^{(h)}, f(\boldsymbol{x}_i^{(h)})\right), \; h = 1, \ldots, H\right\}$ of bee i. The neighborhood search for employed and onlooker bees is thus:

$$\hat{x}_{ij}^{(t+1)} = x_{ij}^{(t)} + \tilde{\phi}(x_{ij}^{(t)} - x_{kj}^{(t)}), \tag{8}$$

where $\tilde{\phi} = \arg\max_{\phi \in [-1,1]} \tilde{f}_i\left(x_{ij}^{(t)} + \phi(x_{ij}^{(t)} - x_{kj}^{(t)})\right)$. In this way, we enhance the exploitation of the discrete domain while preserving the original randomness of the ABC algorithm.

C. Enhancing Exploration via Lévy Flight

One limitation highlighted by the author of the original ABC algorithm [4] is the risk of premature convergence to local optima. To address this issue, a common approach is to enhance the exploration of the algorithm, particularly when the majority of bees are concentrated in a specific region. One possibility is to reduce Λ, the number of improvement trials starting from a specific position after which a bee enters the scout phase. However, this may negatively impact the exploitation of regions where the optimum is believed to lie. A more effective approach is to incorporate Lévy flights into the algorithm [17]. Lévy flights are random walks whose step lengths are drawn from the Lévy distribution [18]. Unlike the steps performed in (2), a Lévy flight is independent of the positions of other bees. As a result, it enhances domain exploration, especially in later iterations, when bees tend to concentrate in specific regions. In our ABC-MLCO algorithm, we introduce this technique during the onlooker phase. Specifically, we define a linearly increasing probability that bee i performs a Lévy flight step, ranging from 10% to 50%:

$$p_i^{\text{Lévy flight}} = 0.1 + 0.4 \cdot \max\left(\frac{\lambda_i}{\Lambda}, 1\right), \tag{9}$$

where λ_i is the number of actual improvement trials of bee i. Further details about the computation of the Lévy step can be found in [18]. This approach enhances the ability of the algorithm to escape local optima and improves exploration.

D. Avoiding repetition in the discrete domain

Since the ABC algorithm was originally developed for optimization in continuous domains [4], handling discrete variables requires particular attention. A naive approach consists of rounding the point obtained during the employed or onlooker phase using (8) to the nearest discrete point within the domain. While valid in its simplicity, this approach carries the risk of repeatedly revisiting points that have already been observed. In our tests, this results in approximately 60% of the visited points being duplicates, uniformly distributed across the iterations. To address this, and given that the bees are equipped with local memory, we propose approximating the point obtained via (8) with the nearest discrete point along the movement direction j that has not yet been visited since bee i entered the initialization or scout phase. If that direction has been exhaustively explored by bee i, the bee selects a different direction in which to move. Should all d movement directions be exhausted, the bee enters the scout phase. This process enhances both the exploitation and exploration capabilities

of the ABC algorithm, reducing the number of repeated evaluations to around 10%.

E. The algorithm

Algorithm 1 outlines the proposed ABC-MLCO approach. Specifically, it takes as input the number of bees N, the number of iterations T, and the maximum number of trial improvements Λ after which a bee becomes a scout. After a random initialization phase (line 2), the ABC iterations begin. Each iteration is split into the *Employed*, *Onlooker*, and *Scout* bee phases. In the Employed phase (lines 4–7), each bee explores a candidate location $\hat{\boldsymbol{x}}_i^{(t)}$ in its neighborhood, leveraging its surrogate ML model $\tilde{f}_i$ trained on its own memory to choose the best location to move to. In the Onlooker phase (lines 8–21), the fitness values $\tilde{fit}_i^{(t)}$, the probability distribution $s^{(t)}$ for the roulette wheel mechanism, and the probability of Lévy flight for each bee are computed. Then, for each bee, a Lévy flight step is performed with probability $p_i^{\text{Lévy flight}}$; otherwise, a bee k is selected via the roulette wheel with probability $s_k^{(t)}$, and bee i explores a location in the neighborhood of $\boldsymbol{x}_k^{(t)}$. Note that in both the Employed and Onlooker phases, a candidate location $\hat{\boldsymbol{x}}_i^{(t)}$ is first explored. Then, $\boldsymbol{x}_i^{(t)}$ is updated to $\hat{\boldsymbol{x}}_i^{(t)}$ if the candidate has a higher fitness value than the current solution (i.e., a lower OF value, with $\boldsymbol{x}$ lying within the feasible domain); otherwise, it remains unchanged. In the former case, the trial counter λ_i is reset to 0; in the latter, λ_i is incremented. Finally, in the Scout phase (lines 22–29), bee i chooses a random point within the feasible domain estimated by a surrogate ML model $\tilde{g}_i$ trained on its history. In this phase, the new position becomes the sampled one. Note that if the new location is infeasible, the bee will continue to perform scout steps in subsequent Employed phases until a feasible location is found.

Algorithm 1 *ABC-MLCO Algorithm*

1: **Input**: *num. of bees* N, *num. of iter.* T, *max trials* Λ
2: **Initialization**: Randomly initialize bees' positions $\boldsymbol{x}_i^{(0)}$ $\forall i$
3: **for** iteration $t = 1$ to T **do**
// Employed Bees Phase
4: Train ML models $\tilde{f}_i$ on local bee i's history, $\forall i$
5: Explore locations $\hat{\boldsymbol{x}}_i^{(t)}$ using (8), $\forall i$
6: Evaluate $f(\hat{\boldsymbol{x}}_i^{(t)})$ and $g(\hat{\boldsymbol{x}}_i^{(t)})$, $\forall i$
7: Update *current best pos.* $\boldsymbol{x}_i^{(t)}$ and *trial counter* λ_i, $\forall i$
// Onlooker Bees Phase
8: Compute fitness values $\tilde{fit}_i^{(t)}$ using (6), $\forall i$
9: Compute selection prob. distribution $s^{(t)}$ using (4), $\forall i$
10: Compute Lévy flight prob. $p_i^{\text{Lévy flight}}$ using (9), $\forall i$
11: **for** bee $i = 1$ to N **do**
12: Sample $l_i \sim$ *Bernoulli*$(p_i^{\text{Lévy flight}})$
13: **if** $l_i = 1$ **then**
14: Explore location $\hat{\boldsymbol{x}}_i^{(t)}$ using a Lévy flight step
15: **else**
16: Sample bee k from the distribution $s^{(t)}$
17: Explore $\hat{\boldsymbol{x}}_i^{(t)}$ using $\boldsymbol{x}_k^{(t)}$ as starting pos. via (8)
18: **end if**
19: Evaluate $f(\hat{\boldsymbol{x}}_i^{(t)})$ and $g(\hat{\boldsymbol{x}}_i^{(t)})$
20: Update *current best pos.* $\boldsymbol{x}_i^{(t)}$ and *trial counter* λ_i
21: **end for**
// Scout Bees Phase
22: **for** bee $i = 1$ to N **do**
23: **if** $\lambda_i \geq \Lambda$ **then**
24: Train ML model $\tilde{g}_i$ on bee i's local history
25: Update position $\boldsymbol{x}_i^{(t)}$ using (7)
26: Evaluate $f(\boldsymbol{x}_i^{(t)})$ and $g(\boldsymbol{x}_i^{(t)})$
27: $\lambda_i \leftarrow 0$
28: **end if**
29: **end for**
30: **end for**
31: **return** Best solution found $\boldsymbol{x}_i^{(T)}$

VI. Experimental Results

In this section, we present the experimental results that validate our ABC-MLCO approach. Section VI-A describes the experimental setup and the evaluation metrics employed in our study. The validation is carried out in two main phases. First, we evaluate the performance of our method on a set of well-known benchmark functions from the literature, appropriately discretized and constrained (Section VI-B). The results are compared with those obtained using the classical ABC algorithm. In the second phase, we apply our approach to a real-world HPC optimization problem, previously addressed in our earlier works [16], [19], namely *LiGen* [2] (Section VI-C). In this case, we also benchmark ABC-MLCO against two alternative methods: OpenTuner [6], a widely used autotuner in the HPC domain, and EMaliboo [5], a parallel Bayesian Optimization (BO) framework specifically designed for the *LiGen* optimization problem.

A. Experimental Setup

The ABC-MLCO algorithm was executed on a Linux server equipped with a 40-core Intel(R) Xeon(R) CPU running at 2.20 GHz and 64 GB of RAM. In the worst-case scenario–namely, the *LiGen* problem–the optimization required approximately 200 seconds using a single core to complete 100 iterations of Algorithm 1. The time per iteration is therefore negligible compared to the evaluation time of the OF, which averages around 9 minutes. To ensure the robustness of the experimental analysis, each experiment was repeated $R = 10$ times using different random seeds, while maintaining the same initialization for both our algorithm and the original ABC algorithm. Regarding the hyperparameters of our method (and the original ABC), we set $N = 50$ and $\Lambda = 10$. The number of iterations was set to $T = 50$ for the test functions and $T = 100$ for the *LiGen* scenario. For the ML models $\tilde{f}_i(\cdot)$ and $\tilde{\boldsymbol{g}}_i(\cdot)$ we employed Ridge regression with second-degree polynomial feature expansion, which provided fast training times (resulting in an overhead of under 0.03 seconds per model training and evaluation in the worst case) and accurate

predictions. Note that the ML models are employed only when the local memory contains at least $H_{\min} = 10$ evaluations of both the OF and the CF, which occurs, in the worst case, starting from the tenth iteration of our algorithm. When $H < H_{\min}$, i.e., during the early iterations, the algorithm performs the same updates as the original ABC algorithm, using (2) instead of (8), and (5) instead of (7). This choice ensures that the ML models are used only when they are sufficiently accurate. On the other hand, we set $H_{\max} = 1000$ to limit the training time of the ML models. If the history exceeds $H_{\max}$, only the $H_{\max}$ most recent evaluations of the OF and CFs are retained. Finally, we present the metrics we use to evaluate different approaches:

- *Mean Absolute Percentage Regret* (MAPR), the average relative difference between the best solution found and the true optimum: $\text{MAPR}(\hat{\boldsymbol{f}}, f^*) = \frac{1}{R}\sum_{r=1}^{R}\left|\frac{\hat{f}_r - f^*}{f^*}\right|$, where $\hat{\boldsymbol{f}}$ is the vector of the best OF values across repetitions, and f^* is the ground-truth minimum. We visualize the variability of this value using box plots.
- *Feasible rate*, the proportion of feasible configurations explored relative to the total number of configurations evaluated. This metric provides insight into the ability of an algorithm to identify feasible regions.

Note that, for a fair comparison with the original ABC algorithm–which addresses unconstrained optimization problems–we also incorporate the fitness function defined in 6 into this approach. The source code for reproducing our analysis, along with our results, are available on Zenodo [1].

B. Test Functions

In the first phase of our experimental campaign, we aim to observe the improvement of our approach over the original ABC framework using well-known test functions from the literature [4], [9]–[11], [15]. Specifically, we test our framework on the *Levy*, *Rastrigin*, *Hartmann*, *Schwefel*, and *Ackley* functions, using dimensionalities ranging from 4 to 8, which correspond to the dimensionality of the *LiGen* scenario, as detailed in Section VI-C. Since these functions are continuous and their domains are well-defined, we define a non-linear black-box constraint for each function, $g(\boldsymbol{x}) \geq 0$, to be satisfied by the final solution. Note that we choose these CFs so that their global optimum f^*, which is evaluated with an exhaustive search in the discrete domain, is higher than that of the corresponding continuous functions. Additionally, we discretize the original continuous domain by randomly selecting $|D_i|$ values within the original range to define a grid-based discrete domain. The analytical formulations of the black-box OFs and CFs, along with their domains, are provided in Table IV (Appendix[1]). Moreover, Table I presents, for each test function, the dimensionality of the domain d, the number of different random values selected for each dimension $|D_i|$, the cardinality of the grid-based domain $|D|$, and the percentage of feasible configurations within the domain relative to $|D|$. Note that the difficulty of the optimization problem increases progressively as the dimensionality grows, both in terms of the cardinality of the domain–exceeding one billion for *Schwefel* and *Ackley*–and in terms of the percentage of feasible configurations, which is 13.69% for *Schwefel*.

TABLE I
OPTIMIZATION DOMAIN FEATURES OF THE TEST FUNCTIONS.

Test function	d	$\lvert D_i\rvert$	$\lvert D\rvert$	Feasible config.
Levy	4	100	1.0e8	38.89 %
Rastrigin	5	60	7.8e8	48.50 %
Hartmann	6	30	7.3e8	20.45 %
Schwefel	7	20	1.3e9	13.69 %
Ackley	8	15	2.6e9	31.82 %

We now present the numerical results. Figure 1 displays the boxplots of the MAPR values obtained by the proposed ABC-MLCO algorithm compared to the standard ABC algorithm. A single orange line indicates that the entire distribution of MAPRs is tightly concentrated around the median. The figure

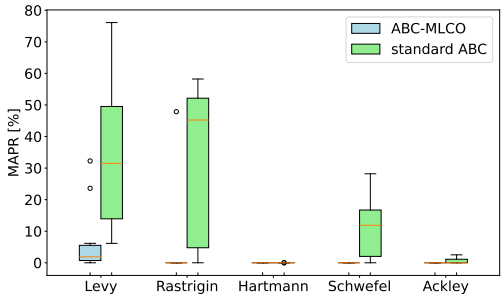

Fig. 1. Boxplots of the MAPRs achieved by the proposed ABC-MLCO algorithm and the standard ABC algorithm on the test functions.

shows that the proposed approach outperforms the standard ABC algorithm across all test functions. This result is further highlighted in Table II. Specifically, it reports the final MAPR after $T = 50$ iterations, for each test function and averaged over $R = 10$ repetitions; the iteration at which the MAPR falls below 10%, 5%, and 1%; and the feasibility rate for all function-algorithm pairs. These results demonstrate that the

TABLE II
EXPERIMENTAL RESULTS ON THE TEST FUNCTIONS: MAPR, ITERATION TO REACH K% OF MAPR, AND FEASIBILITY RATE.

Function	Algorithm	MAPR	Iter. MAPR $\leq$ 10%	5%	1%	Feas. rate
Levy	*ABC-MLCO*	7.16	48	-	-	75.22 %
	Std ABC	34.06	-	-	-	73.22 %
Rastrigin	*ABC-MLCO*	4.79	41	49	-	77.20 %
	Std ABC	32.68	-	-	-	71.24 %
Hartmann	*ABC-MLCO*	0.00	2	3	7	72.85 %
	Std ABC	0.00	5	8	23	59.04 %
Schwefel	*ABC-MLCO*	0.00	10	15	26	63.09 %
	Std ABC	11.24	-	-	-	45.16 %
Ackley	*ABC-MLCO*	0.00	3	5	9	75.38 %
	Std ABC	0.66	13	18	44	54.08 %

MAPR of ABC-MLCO is not only lower than that of the original ABC at the final iteration–particularly for the *Levy* and *Rastrigin* problems (by -27%)–but also that our algorithm reduces it more quickly, highlighting improved exploitation ability. Moreover, ABC-MLCO more effectively defines the feasible region, achieving an increased feasible rate of around 20% in the *Schwefel* and *Ackley* cases, which reflects its enhanced exploration capability. Note that the ability to explore

[1]https://doi.org/10.5281/zenodo.16987523

feasible configurations directly leads to reduced optimization costs, particularly in the context discussed in the next section. Finally, Figure 2 illustrates, as a representative example, the monotonically decreasing MAPR trends for both the ABC-MLCO and standard ABC algorithms in the *Rastrigin* case. As shown in Table II, the MAPR of ABC-MLCO drops below 10% at iteration 41 and below 5% at iteration 49, while the standard ABC never falls below 30%.

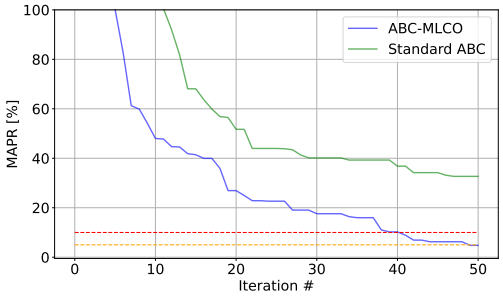

Fig. 2. MAPR trends of ABC-MLCO and the standard ABC algorithms over the iterations on *Rastrigin*. The dashed red and orange lines depict, respectively, 10% and 5% of the MAPR.

C. The LiGen Application

LiGen is a molecular docking application integrated into the EXSCALATE drug discovery platform [2]. It estimates how well a ligand, i.e. a drug candidate, interacts with a target protein. First, it determines the 3D displacement of the ligand using multiple descent-gradient restarts followed by cluster analysis. Then, it computes the interaction strength using a scoring function [20]. The quality of the docking solution is assessed by calculating the average Root Mean Square Deviation (RMSD) across 100 ligand-protein pairs with experimental data. In this context, each configuration $\boldsymbol{x}$ is defined by eight discrete parameters. Fine-tuning these parameters is crucial for balancing performance and accuracy, a particularly challenging task due to the vast parameter space, which includes approximately half a million configurations. The application outputs include the RMSD $R(\boldsymbol{x})$ and execution time $T(\boldsymbol{x})$. The OF to minimize is $f(\boldsymbol{x}) = R^3(\boldsymbol{x})T(\boldsymbol{x})$, subject to a quality constraint $R(\boldsymbol{x}) \leq R_{\max}$, where we consider $R_{\max} = \{2.0, 2.1, 2.2, 2.45, 2.75\}$ as the RMSD thresholds deemed relevant by domain experts.

We now present two other approaches with which we compare our ABC-MLCO, followed by the numerical results.

1) Alternative Approaches: We introduce *EMaliboo* and *OpenTuner*, two optimization algorithms from the literature that we use for comparison in the optimization of *LiGen*.

EMaliboo [5] (Ensemble MAchine Learning In Bayesian OptimizatiOn) is an extension of the MALIBOO algorithm [16]. MALIBOO is a BO algorithm specifically designed to address constrained optimization problems. It integrates ML into the BO framework to model the relationship between input configurations and black-box constraints. Assuming that N *parallel workers* are available, EMaliboo employs an ensemble of N independent agents, each executing its own instance of the sequential MALIBOO algorithm. The authors of [5] also propose a purely asynchronous parallel extension of MALIBOO, where a centralized agent sequentially selects new configurations and immediately assigns each to a parallel worker for evaluation. However, in the context of *LiGen*, the EMaliboo approach proved more effective, as its ensemble of independent agents achieved better exploration of the configuration space. For this reason, we compare our ABC-MLCO approach exclusively against EMaliboo.

OpenTuner [6] is a widely adopted autotuner that uses an ensemble of search techniques to explore large and complex configuration spaces. Each technique independently evaluates candidate configurations and contributes to a shared database of results. Techniques that consistently identify high-performing configurations are allocated more computational resources. OpenTuner manages this resource allocation dynamically by addressing the multi-armed bandit problem, using the area under the curve as its credit assignment strategy.

Note that for both approaches, we conducted experiments using $N = 50$ (matching the number of bees in our ABC-MLCO) and $N = 10$. Since the latter yielded better performance for both methods, we report results only for $N = 10$.

2) Numerical results: We now present the numerical results for the *LiGen* scenario. Specifically, Table III reports the MAPR and feasibility rate for each value of the black-box constraint, while Figure 3 shows boxplots of the MAPR values obtained across the $R = 10$ repetitions of each experiment.

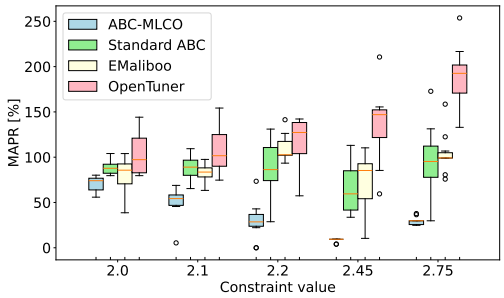

Fig. 3. Boxplots of the MAPRs achieved by ABC-MLCO, standard ABC, EMaliboo and OpenTuner algorithms on *LiGen*.

Our proposed ABC-MLCO algorithm outperforms all competing methods in terms of MAPR. Specifically, the standard ABC algorithm yields MAPR values that are 18–68% worse, EMaliboo performs 11–80% worse, and OpenTuner shows a 34–160% degradation in MAPR compared to ABC-MLCO. Regarding the feasibility rate, EMaliboo performs best on average. The largest gap appears at a constraint value of 2.1, where it outperforms ABC-MLCO by approximately 23.5%. However, under the strictest constraint, ABC-MLCO achieves a 6% higher feasibility rate. Considering the average MAPR gap of 52.4%, EMaliboo appears significantly less explorative than our ABC-MLCO. The difference in feasibility rate can be attributed to two factors. First, the ML models in EMaliboo have broader domain coverage, in contrast to the *local* information used by the bees in ABC-MLCO, whose movements remain localized until entering the scout phase. Second, BO-based algorithms are more exploitation-focused than ABC,

TABLE III
EXPERIMENTAL RESULTS ON *LiGen*: MAPR AND FEASIBILITY RATE. IN **BOLD** THE BEST VALUES.

Algorithm	$R(x) \leq 2.0$		$R(x) \leq 2.1$		$R(x) \leq 2.2$		$R(x) \leq 2.45$		$R(x) \leq 2.75$	
	MAPR	Feas. rate	MAPR	Feas. rate	MAPR	Feas. rate	MAPR	Feas. rate	MAPR	Feas. rate
ABC-MLCO	**70.23**	**28.24 %**	**50.46**	50.03 %	**29.46**	60.35 %	**8.34**	82.86 %	**29.60**	92.22 %
Standard ABC	88.68	12.54 %	87.76	22.20 %	87.73	26.64 %	64.53	33.78 %	97.19	35.57 %
EMaliboo	81.34	22.18 %	81.95	**73.54 %**	109.10	**77.39 %**	73.66	**93.21 %**	104.00	**99.58 %**
OpenTuner	103.83	6.99 %	107.35	11.29 %	116.45	22.08 %	135.71	20.19 %	189.26	22.35 %

so when EMaliboo finds a feasible region, it requires more iterations to escape local optima, which increases the number of feasible configurations evaluated. By contrast, both the standard ABC algorithm and OpenTuner achieve significantly worse results in terms of feasibility rate as well, compared to ABC-MLCO. Finally, Figure 4 illustrates the faster decrease in MAPR achieved by ABC-MLCO compared to all other approaches in the scenario of $R(\boldsymbol{x}) \leq 2.1$.

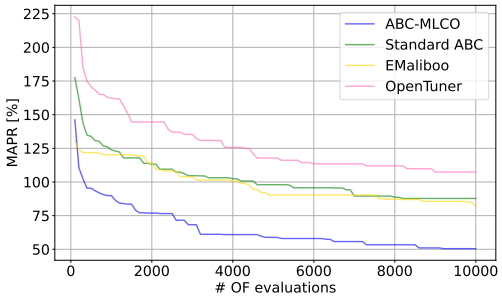

Fig. 4. MAPR trends of ABC-MLCO, standard ABC, EMaliboo and OpenTuner algorithms on *LiGen*, with constraint $R(\boldsymbol{x}) \leq 2.1$.

For the sake of space, detailed comparisons of total execution time (OF evaluation plus algorithmic iterations) are not reported. In *LiGen*, all competitor approaches exhibit longer execution times than ABC-MLCO: for EMaliboo this stems from expensive model updates, whereas for standard ABC and OpenTuner it results from less efficient search leading to slower OF evaluations.

VII. CONCLUSIONS AND FUTURE WORK

This work introduced ABC-MLCO, an ABC-based algorithm enhanced with ML to address constrained optimization in discrete domains, in particular in the field of HPC. Compared to the original ABC framework, we incorporate techniques for adapting the original algorithm to the proposed scenario–exploiting ML models to define the feasible region and guide the search for the optimum–and improving algorithm efficiency through a local memory that avoids redundant evaluations of the OF and supplies training data for the ML models. Experimental results demonstrate improvements in regret, feasibility rate, and convergence speed compared to the original ABC algorithm on benchmark functions. Moreover, ABC-MLCO outperforms state-of-the-art approaches such as EMaliboo and OpenTuner, achieving average regret reductions of 52% and 93%, respectively, in autotuning the *LiGen* application. Future work will involve an ablation study of the parameters of ABC-MLCO, including the choice of ML models, as well as a sensitivity analysis of each component introduced. Furthermore, we will evaluate our algorithm on a broader range of real-world applications. Finally, our work will focus on developing a fully asynchronous version of ABC-MLCO to better utilize available parallel resources, and on introducing mixed strategies–such as BO–to further enhance the exploitation capabilities of the algorithm.

REFERENCES

[1] J. Glaser *et al.*, "High-throughput virtual laboratory for drug discovery using massive datasets," *The International Journal of High Performance Computing Applications*, vol. 35, no. 5, pp. 452–468, 2021.

[2] D. Gadioli *et al.*, "Exscalate: An extreme-scale virtual screening platform for drug discovery targeting polypharmacology to fight sars-cov-2," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 1, pp. 170–181, 2022.

[3] Y. Tan *et al.*, "A survey on gpu-based implementation of swarm intelligence algorithms," *IEEE transactions on cybernetics*, vol. 46, no. 9, pp. 2028–2041, 2015.

[4] D. Karaboga *et al.*, "A comparative study of artificial bee colony algorithm," *Applied mathematics and computation*, vol. 214, no. 1, pp. 108–132, 2009.

[5] B. Guindani *et al.*, "Efficient parameter tuning for a structure-based virtual screening hpc application," *Journal of Parallel and Distributed Computing*, vol. 202, p. 105 087, 2025.

[6] J. Ansel *et al.*, "Opentuner: An extensible framework for program autotuning," in *IEEE PACT*, 2014, pp. 303–316.

[7] V. Tereshko, "Reaction-diffusion model of a honeybee colony's foraging behaviour," in *PPSN*, Springer, 2000, pp. 807–816.

[8] V. Tereshko *et al.*, "Collective decision making in honey-bee foraging dynamics," *Computing and information systems*, vol. 9, no. 3, p. 1, 2005.

[9] W.-F. Gao *et al.*, "Artificial bee colony algorithm based on information learning," *IEEE transactions on cybernetics*, vol. 45, no. 12, pp. 2827–2839, 2015.

[10] M. S. Kıran *et al.*, "A directed artificial bee colony algorithm," *Applied Soft Computing*, vol. 26, pp. 454–462, 2015.

[11] H. Wang *et al.*, "Improving artificial bee colony algorithm using a new neighborhood selection mechanism," *Information Sciences*, vol. 527, pp. 227–240, 2020.

[12] M. Babar *et al.*, "Intelligent computation offloading for iot applications in scalable edge computing using artificial bee colony optimization," *Complexity*, vol. 2021, no. 1, p. 5 563 531, 2021.

[13] W. Liu *et al.*, "Fog computing resource-scheduling strategy in iot based on artificial bee colony algorithm," *Electronics*, vol. 12, no. 7, p. 1511, 2023.

[14] S. S. Nabavi *et al.*, "Tractor: Traffic-aware and power-efficient virtual machine placement in edge-cloud data centers using artificial bee colony optimization," *International Journal of Communication Systems*, vol. 35, no. 1, e4747, 2022.

[15] X. Li *et al.*, "Artificial bee colony algorithm with memory," *Applied Soft Computing*, vol. 41, pp. 362–372, 2016.

[16] B. Guindani *et al.*, "Integrating bayesian optimization and machine learning for the optimal configuration of cloud systems," *IEEE transactions on cloud computing*, vol. 12, no. 1, pp. 277–294, 2024.

[17] G. M. Viswanathan *et al.*, "Lévy flight search patterns of wandering albatrosses," *Nature*, vol. 381, no. 6581, pp. 413–415, 1996.

[18] X.-S. Yang, *Nature-inspired metaheuristic algorithms*. Luniver press, 2010.

[19] R. Sala *et al.*, "D-MALIBOO: A bayesian optimization framework for dealing with discrete variables," in *MASCOTS*, IEEE, 2024, pp. 1–8.

[20] E. Vitali *et al.*, "Exploiting openmp and openacc to accelerate a geometric approach to molecular docking in heterogeneous hpc nodes," *The Journal of Supercomputing*, vol. 75, pp. 3374–3396, 2019.

TABLE IV
ANALYTICAL DEFINITIONS OF TEST FUNCTIONS AND CONSTRAINTS.

Function	$f(\mathbf{x})$ and $g(\mathbf{x})$	**Range of definition**
Levy	$f(\mathbf{x}) = \sin^2(\pi w_1) + \sum_{i=1}^{d-1} \left[(w_i - 1)^2 \cdot \left(1 + 10 \cdot \sin^2(\pi w_i + 1)\right)\right] + (w_d - 1)^2 \cdot \left(1 + \sin^2(2\pi w_d)\right)$, $w_i = 1 + \frac{x_i - 1}{4}$, $i = 1, \ldots, d$, $\quad g(\mathbf{x}) = \log(x_1^2 + x_3^2) - (x_2 + x_4)^3 - 4$	$x_i \in [-10, 10]\ \forall i$
Rastrigin	$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left(x_i^2 - 10\cos(2\pi x_i)\right)$ $g(\mathbf{x}) = 2 - d \cdot \exp\left(\sum_{i=1}^{d} x_i\right)$	$x_i \in [-5.12, 5.12]\ \forall i$
Hartmann	$f(\mathbf{x}) = -\sum_{i=1}^{4} \alpha_i \exp\left(-\sum_{j=1}^{6} A_{ij}(x_j - P_{ij})^2\right)$, $A = \begin{bmatrix} 10 & 3 & 17 & 3.5 & 1.7 & 8 \\ 0.05 & 10 & 17 & 0.1 & 8 & 14 \\ 3 & 3.5 & 1.7 & 10 & 17 & 8 \\ 17 & 8 & 0.05 & 10 & 0.1 & 14 \end{bmatrix}$, $P = 10^{-4} \begin{bmatrix} 1312 & 1696 & 5569 & 124 & 8283 & 5886 \\ 2329 & 4135 & 8307 & 3736 & 1004 & 9991 \\ 2348 & 1451 & 3522 & 2883 & 3047 & 6650 \\ 4047 & 8828 & 8732 & 5743 & 1091 & 381 \end{bmatrix}$, $\alpha = \begin{bmatrix} 1.0 & 1.2 & 3.0 & 3.2 \end{bmatrix}$, $\quad g(\mathbf{x}) = \frac{4\left(x_1^2 + x_4^3\right)^{1/8}}{x_2 x_3 - \sinh(x_5)} - 1$	$x_i \in [0, 1]\ \forall i$
Schwefel	$f(\mathbf{x}) = 418.9829 \cdot d - \sum_{i=1}^{d} x_i \cdot \sin\left(\sqrt{\lvert x_i \rvert}\right)$ $g(\mathbf{x}) = \min\Big(\max\left(400^2 - (x_1 - 350)^2 - (x_4 + 150)^2 - (x_7 - 200)^2,\right.$ $\left.300^2 - (x_2 + 100)^2 - (x_5 - 50)^2 - (x_6 + 35)^2\right), \quad -0.001 \cdot x_3^2 + 90\Big)$	$x_i \in [-500, 500]\ \forall i$
Ackley	$f(\mathbf{x}) = -a \cdot \exp\left(-b \cdot \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d} \cos(c x_i)\right) + a + \exp(1)$, $a = 20,\ b = 0.2,\ c = 2\pi, \quad g(\mathbf{x}) = (x_1 - 3x_4)^3 - 2\sqrt{x_8^2 + e^{x_7}} + 5x_5 x_6 + \sin\left(x_2^2 + \cos(x_3)\right) - 2$	$x_i \in [-32.768, 32.768]\ \forall i$

APPENDIX

In this Appendix, Table IV presents the analytical formulations of the black-box OFs and CFs, as well as the original domains of the test functions used to compare ABC-MLCO with the original ABC algorithm, as discussed in Section VI-B. The discrete domain points and global optima f^* are available in the scripts of the ABC-MLCO algorithm on Zenodo ([https://doi.org/10.5281/zenodo.15345318).