

# EvoX: A Distributed GPU-Accelerated Framework for Scalable Evolutionary Computation

Beichen Huang<sup>1</sup>, Graduate Student Member, IEEE, Ran Cheng<sup>2</sup>, Senior Member, IEEE,  
Zhuzhao Li<sup>1</sup>, Yaochu Jin<sup>2</sup>, Fellow, IEEE, and Kay Chen Tan<sup>3</sup>, Fellow, IEEE

**Abstract**—Inspired by natural evolutionary processes, evolutionary computation (EC) has established itself as a cornerstone of artificial intelligence. Recently, with the surge in data-intensive applications and large-scale complex systems, the demand for scalable EC solutions has grown significantly. However, most existing EC infrastructures fall short of catering to the heightened demands of large-scale problem solving. While the advent of some pioneering GPU-accelerated EC libraries is a step forward, they also grapple with some limitations, particularly in terms of flexibility and architectural robustness. In response, we introduce **EvoX**: a computing framework tailored for automated, distributed, and heterogeneous execution of EC algorithms. At the core of **EvoX** lies a unique programming model to streamline the development of parallelizable EC algorithms, complemented by a computation model specifically optimized for distributed GPU acceleration. Building upon this foundation, we have crafted an extensive library comprising a wide spectrum of 50+ EC algorithms for both single- and multi-objective optimization. Furthermore, the library offers comprehensive support for a diverse set of benchmark problems, ranging from dozens of numerical test functions to hundreds of reinforcement learning tasks. Through extensive experiments across a range of problem scenarios and hardware configurations, **EvoX** demonstrates robust system and model performances. **EvoX** is open source and accessible at: <https://github.com/EMI-Group/EvoX>.

**Index Terms**—Distributed computing, evolutionary reinforcement learning, GPU acceleration, neuroevolution, scalable evolutionary computation (EC).

Manuscript received 6 October 2023; revised 6 February 2024; accepted 8 April 2024. Date of publication 15 April 2024; date of current version 8 October 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62202216; in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2023A1515010244; and in part by the Shenzhen Science and Technology Program under Grant 20231121101752002. This article was approved by Associate Editor A. K. Qin. (Corresponding author: Ran Cheng.)

Beichen Huang was with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China. He is now with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, SAR (e-mail: beichen.huang@connect.polyu.hk).

Ran Cheng and Zhuzhao Li are with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: ranchengcn@gmail.com; lizz@sustech.edu.cn).

Yaochu Jin is with the School of Engineering, Westlake University, Hangzhou 310030, China (e-mail: jinyaochu@westlake.edu.cn).

Kay Chen Tan is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, SAR (e-mail: kctan@polyu.edu.hk).

Digital Object Identifier 10.1109/TEVC.2024.3388550

## I. INTRODUCTION

INSPIRED by the process of natural evolution, evolutionary computation (EC) has established its significance as a distinctive discipline within the expansive realm of artificial intelligence (AI) [1]. EC's pivotal role is underscored by its inherent attributes: adaptability, resilience, and aptitude, which are pivotal for complex problem solving [2], [3], [4]. Furthermore, EC plays a pivotal role in the quest for Artificial General Intelligence, especially within the context of neuroevolution [5].

In the contemporary era, marked by large volumes of data and complex systems across diverse domains, the emphasis on *scalability* in EC has gained paramount importance [6], [7]. As problem complexity and dimensionality increase, especially in fields like deep learning, there is an increasing demand for EC infrastructures to accommodate larger population sizes and higher-dimensional problems [8], [9]. Looking ahead, scalable EC will not only facilitate tackling larger problems but also pave the way toward emulating the complexity and adaptability inherent in biological systems [10]. However, most existing EC infrastructures, including algorithm designs and computing frameworks, were merely tailored for smaller scales. Thus, addressing the challenge of scalability is a critical research frontier in the ongoing development of EC.

Inherently, EC algorithms are well suited to parallel computation due to their usage of a population of candidate solutions, each capable of independent evaluation. Consequently, they stand to benefit substantially from the parallel computation capabilities of hardware accelerators. Traditional CPU-based parallelization remains the prevailing approach, exemplified by DEAP [11], PyGAD [12], Pymoo [13], and Pagmo [14]. Until very recently, some pioneering advancements have led to the emergence of GPU-accelerated EC libraries, such as EvoJAX [15], evosax [16], and EvoTorch [17]. Particularly, EvoJAX and evosax are based on JAX [18], while EvoTorch is built upon PyTorch [19] and Ray [20].

Nonetheless, the pace of integrating hardware accelerators into EC infrastructures has been substantially slower compared to the strides made in the deep learning sector. A primary reason for this discrepancy is the lack of a universally endorsed and scalable computing framework. Existing EC libraries, notably the recent GPU-accelerated ones, offer a plethora of useful tools and features. However, they are not without limitations. EvoJAX and evosax, in spite of harnessing GPU acceleration features, predominantly cater to evolution

strategies with a focus on single-objective optimization, thus limiting their broader applicability. EvoTorch stands out with its adeptness in auto-parallelizing EC algorithms across multiple GPUs. However, its reliance on PyTorch, which is primarily tailored for deep learning rather than scientific computing, could impede its computational efficiency. Additionally, the parallelism of EvoTorch-based solutions is tightly coupled with the operators supported by PyTorch, potentially limiting the flexibility. Most importantly, while EvoJAX, evosax, and EvoTorch are commendable as algorithm libraries, their architectural designs at the framework level are very limited. For example, they lack unified programming and computational models to enable simple implementation of general EC algorithms for seamless execution in distributed environments.

To address these limitations, we introduce *EvoX*, a framework that facilitates automated, distributed, and heterogeneous execution of general EC algorithms. *EvoX* implements a straightforward functional programming model, enabling users to declare the logical flow of an EC algorithm via a unified interface. Moreover, *EvoX* adopts a hierarchical state management strategy that automatically distributes the tasks to arbitrary heterogeneous resources using a distributed execution engine. In summary, the main contributions are as follows.

- 1) We have designed and implemented *EvoX*, a scalable and efficient framework that enables the execution of general EC algorithms across distributed heterogeneous systems.
- 2) We have proposed a straightforward functional programming model within *EvoX* to streamline the development process of general EC algorithms for parallelization. This model allows users to easily declare the logical flow of an EC algorithm, which reduces the complexity typically associated with the development process.
- 3) We have unified the main data stream and functional components into a flexible workflow. This unification is achieved through a hierarchical state management module, which supports high-performance executions of general EC algorithms.
- 4) Leveraging the *EvoX* framework, we have crafted a library that encompasses a wide spectrum of 50+ EC algorithms for both single- and multi-objective optimization. Furthermore, the library has featured intuitive interfaces to diverse benchmark environments, comprehensively supporting hundreds of instances ranging from numerical optimization functions to reinforcement learning tasks.

The remainder of this article is organized as follows. Section II presents some related work. Section III illustrates the motivation and requirements. Section IV details the programming and computation models. Sections V and VI elaborate on the architecture and implementation of *EvoX*, respectively. Section VII conducts the experiments to assess the performance *EvoX*, in comparison with EvoTorch. Finally, Section VIII concludes this article and discusses future work.

## II. RELATED WORK

### A. EC Libraries in Python

*DEAP* [11] stands as a comprehensive framework for EC algorithms in Python. With a rich history and a plethora of features, it caters to a broad spectrum of EC algorithms, encompassing both single- and multi-objective variants. Its collection of built-in benchmark problems facilitates the comprehensive evaluations of EC algorithms.

*PyGAD* [12] is a specialized platform for devising genetic algorithms in Python. It empowers users with a variety of crossover, mutation, and parent selection operators. Notably, it is particularly tailored for machine learning tasks, offering specialized tools and features for neural network training, cementing its role in integrating EC with machine learning pursuits.

*Pymoo* [13] focuses primarily on multiobjective optimization problems, providing a robust platform for EC in Python. Its extensive support for diverse benchmark problems and state-of-the-art multiobjective EC algorithms, combined with visualization tools, highlights its commitment to the EC domain.

*Pygmo* [14], distinguished by its emphasis on massively parallel optimization, adopts the generalized island model for coarse-grained parallelization. It offers a vast array of algorithms and benchmark problems, facilitating the efficient deployment of parallelized EC algorithms. Its batch fitness evaluation feature further enhances its utility.

*EvoJAX* [15] pushes the boundaries of scalable, hardware-accelerated neuroevolution. Leveraging the capabilities of the JAX framework, it seamlessly integrates evolution strategies with neural networks, ensuring efficient GPU parallelism. Building upon JAX, it provides a NumPy-like environment with just-in-time (JIT) compilation.

*evosax* [16], a recent addition, positions itself as a dedicated library for GPU-accelerated evolution strategy (ES). Taking cues from EvoJAX and deeply integrated with the JAX infrastructure, it presents a curated suite of ES algorithms, each of which is optimized for GPU performance.

*EvoTorch* [17] is another emerging library placing an emphasis on addressing scalability challenges within EC by harnessing the power of GPU acceleration. By seamlessly integrating with PyTorch, it not only gains the inherent advantages of this popular deep learning framework but also naturally taps into the vast resources of the Python community.

### B. JAX

JAX [18] has rapidly ascended the ranks to become a leading computational infrastructure. While it offers a NumPy-style API, JAX has been optimized for GPU-accelerated numerical operations. JAX's unique selling point is its adaptability to contemporary computational challenges. Its JIT compilation transforms user-defined Python functions into high-performance machine code adaptable to diverse hardware platforms.

One of JAX's notable features is its ability to fuse operators, thereby amalgamating multiple smaller tensor operations into singular and efficient tasks, thus reducing memory overhead.

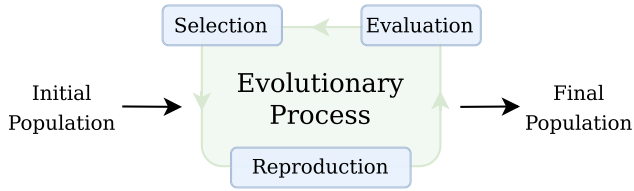


Fig. 1. Typical process of an EC algorithm. Starting with an initial population, the EC algorithm engages in problem solving through an iterative evolutionary process. Specifically, the main loop of this process evolves the population via three primary components: reproduction, evaluation, and selection. Ultimately, the final population is output as the solution set to the problem at hand.

Its integrated autograd system enables automatic gradient computations, an essential feature for gradient-based optimization. Adhering to functional programming paradigms, JAX ensures computations are pure and side-effect-free, resulting in predictable and debug-friendly code. Such principles align well with EC algorithms, which are stateless by nature and suitable for a functional setting.

### III. MOTIVATION AND REQUIREMENTS

As shown in Fig. 1, the typical process of an EC algorithm involves evolving a *population* of candidate solutions to a specific problem. In each iteration of the main loop, the current population first undergoes a *reproduction* phase to generate new candidate solutions. Next, the *evaluation* phase assesses the effectiveness, also known as *fitness*, of each candidate solution in solving the problem at hand. Following evaluation, a *selection* process is carried out, favoring candidate solutions with superior fitness for inclusion in the subsequent generation, in line with the principle of *survival of the fittest*. This process repeats until either a satisfactory solution or set of solutions is found, or a predetermined number of iterations are completed.

From the perspective of a computing framework, the *population* can be viewed as the primary data flow, while the *reproduction*, *evaluation*, and *selection* are three functional components. Therefore, a computing framework designed for scalable EC must efficiently support these workloads. We provide a brief description of these workloads below.

- 1) *Population*: This object, consisting of candidate solutions in various data structures, goes through each functional component of the entire evolutionary process. In some emerging applications of EC such as deep neuroevolution [5], the population may require computationally expensive decoding for complex representations.
- 2) *Reproduction*: This process, such as the crossover/mutation operators in a genetic algorithm, often consists of a set of heuristic strategies for generating new candidate solutions. Reproduction can be parallelized in most EC algorithms that adopt dimension-wise operations when generating each new candidate solution.
- 3) *Evaluation*: This process can be intrinsically parallelized in a distributed setting as the evaluation of each candidate solution is independent. In some emerging applications of EC such as evolutionary reinforcement learning [21], the computing task may require heterogeneous hardware (i.e., CPUs and GPUs) for hybrid complex simulations and deep learning tasks.

- 4) *Selection*: This process is often realized via a set of sorting/ranking strategies, which can be compute-intensive with respect to population size. Parallelized selection is particularly beneficial in EC paradigms involving multiple populations.

Unlike deep learning, which benefits from a streamlined end-to-end workflow, the various components of an EC workflow, including data flow processing and functional elements, play distinct yet interconnected roles. This uniqueness poses significant challenges when designing a general computing framework to support EC algorithms. Moreover, as the applications of EC continue to expand, a widening gap has emerged between EC and other major branches of AI. Bridging this gap necessitates the development of a distributed computing framework that can efficiently handle EC workloads, ensuring scalability and compatibility with heterogeneous computing environments. Specifically, such a framework should meet the following requirements.

- 1) *Flexibility and Extensibility*: The framework should support easy implementation of a broad spectrum of EC algorithms and black-box optimization problems. To this end, it should provide flexibility in the representation of candidate solutions (e.g., strings, trees, and neural networks), the combination of various reproduction and selection mechanisms, as well as diverse methods for evaluation. Users should be able to leverage the framework to integrate existing modules seamlessly, thereby constructing workflows that align with their specific research goals and problem contexts.
- 2) *Parallelizable Programming*: The framework should adopt a programming model geared toward parallelization. While this model offers advantages, such as improved code clarity, testability, and reusability, its primary strength lies in its innate ability to support parallel execution. Consequently, even EC algorithms developed without direct parallel considerations should intuitively harness this capability, thus allowing for efficient execution across multiple computing units concurrently.
- 3) *Heterogeneous Execution*: The framework should enable task executions across various types of computing resources, including CPUs, GPUs, or even other specialized hardware. It should also support distributed computing across multiple nodes, which involves intelligent data sharding, task scheduling, and resource management based on the requirements of tasks and available hardware. It should transparently manage data transfer and synchronization between different nodes.

In response to these requirements, we design and implement EvoX. In the following section, we will elaborate on the programming and computation models underpinning this framework.

### IV. PROGRAMMING AND COMPUTATION MODELS

#### A. Programming Model

To meet the requirement of *Parallelizable Programming* (outlined in Section III), we adopt the *functional programming* paradigm as the core of our programming model. This

```

1 from evox import Algorithm, State
2
3 class EC(Algorithm):
4     def __init__(self, reproduction, selection, ...):
5         # initialize operators
6         super().__init__()
7         self.reproduce = reproduction
8         self.select = selection
9         ...
10
11     def setup(self, key):
12         # initialize data
13         init_population = ...
14         init_fitness = ...
15         ...
16         # initialize state
17         return State(
18             pop = init_population,
19             fit = init_fitness,
20             ...
21             key=key
22         )
23
24     def ask(self, state):
25         # generate offspring
26         offspring = self.reproduce(state.pop, ...)
27         # update state
28         state = state.update(
29             # merge offspring into the population
30             pop = merge(pop, offspring),
31             ...
32         )
33         return offspring, state
34
35     def tell(self, state, fitness):
36         # select new population
37         new_pop, new_fit = self.select(
38             state.pop, merge(state.fit, fitness))
39         # update state
40         state = state.update(
41             pop = new_pop,
42             fit = new_fit,
43             ...
44         )
45         return state

```

Listing 1. Exemplar implementation of a vanilla EC algorithm within the EvoX framework. This implementation comprises four sections: `__init__`, `setup`, `ask`, and `tell`. The `__init__` section is dedicated to initializing foundational operators, including reproduction and selection mechanisms. The `setup` segment focuses on data initialization, encompassing elements like the population data and fitness data. Within the `ask` section, the offspring are produced utilizing the reproduction operator, while the `tell` section facilitates population updates through the selection operator

design is rooted in the paradigm's intrinsic benefits, including streamlined code, augmented reusability and testability, and a natural affinity for parallel and distributed computing. With this model, our aim is to provide an intuitive interface that facilitates the straightforward implementation of a wide spectrum of EC algorithms and their corresponding workflows.

An illustrative example of using the proposed programming model to implement a vanilla EC algorithm is presented in Listing 1. This implementation comprises four sections: `__init__`, `setup`, `ask`, and `tell`. Within the `__init__` section, basic operators essential to the EC algorithm, encompassing processes like reproduction and selection, are defined. Given their invariant nature throughout the evolutionary trajectory, these operators are archived under the Python attribute `self`. The `setup` section is devoted to initializing essential data elements, such as the population data and fitness data,

TABLE I  
SUMMARY OF CORE MODULES AND FUNCTIONS IN  
EVOX'S PROGRAMMING MODEL

Module	Function	Description
Algorithm	ask	To generate offspring population.
	tell	To select new population.
Problem	evaluate	To evaluate the fitness of a given population.
Monitor	record_pop	To record the population data.
	record_fit	To record the fitness data.
Workflow	init	To initialize the workflow.
	step	To execute one iteration of the workflow.

as well as a key for random number generation. As these elements possess a mutable characteristic and are subject to modifications with each iteration, they are aptly demarcated as part of the state. Subsequently, the `ask` section is tailored for the generation of offspring, realized through the designated reproduction operator (e.g., crossover and mutation). In contrast, the `tell` section orchestrates the assimilation of offspring into the prevailing population, culminating in an updated population based on the predefined selection operator. Table I provides a concise overview of the primary modules and functions within the programming model, which are detailed as follows.

*Algorithm* is tailored for encapsulating EC algorithms. At its core is the `ask-and-tell` interface, which conceptualizes the EC algorithm as an agent in perpetual interaction with a problem at hand, involving `ask` and `tell` functions.

- 1) *ask*: It processes the state to yield new candidate solutions, updating the algorithmic state by merging these solutions with the existing population.
- 2) *tell*: After ingesting the state and the fitness metrics of the new offspring, it selects candidate solutions for the next iteration, resulting in a refreshed algorithmic state.

*Problem* is tailored for modeling the problems to be solved. Central to this module is the `evaluate` function dedicated to determining the fitness values of candidate solutions within a population. Notably, the *Problem* module is constructed as a stateful procedure, such that it is capable of facilitating the management of intricate external environments. For instance, neuroevolution tasks reliant on external datasets can use the *Problem* module to manage the current batch via its state.

*Monitor* serves as an optional module for monitoring the data flow when running an EC algorithm. Utilizing callback functions, the *Monitor* module receives user-specified data by employing callback function, e.g., `record_pop` and `record_fit` for population data and fitness data, respectively. Within the *Monitor* module, users can further process the data by statistical means or visualization tools.

*Workflow* plays a pivotal role in integrating various components to shape a cohesive and executable EC workflow. It begins by amalgamating diverse modules of *Algorithm*, *Problem*, and *Monitor* specified by users. Following this, it acts as the primary module, sequentially initiating every component via the `init` function. This orchestration produces



```

1 from evox.workflow import StdWorkflow
2
3 # workflow creation
4 workflow = StdWorkflow(
5     algorithm,
6     problem,
7     monitor
8 )
9
10 # workflow initialization
11 key = ... # A random number generator key
12 state = workflow.init(key)
13
14 # workflow analyzer (optional)
15 state = workflow.enable_multi_devices(state)
16 state = workflow.enable_distributed(state)
17
18
19 while ...: # not terminated
20     # workflow runner
21     state = workflow.step(state) # one iteration

```

Listing 2. Illustrative usage of a workflow in EvoX. First, a workflow object is created. Second, all components within a workflow are initialized, and a global state is returned. Then, the workflow can be configured with the help of a workflow analyzer. Finally, the workflow is executed inside a loop in a distributed environment.

a global state, enveloping individual states as defined by each module's setup method. Listing 2 showcases the usage of the workflow within EvoX. After selecting an algorithm, a problem, and a monitor, users combine these instances within a workflow object. Post-initialization via `init`, which activates every module under the workflow object recursively, users can fine-tune the workflow for advanced computational tasks, thus facilitating the efficient execution of the workflow across multiple nodes. The workflow's execution is then triggered using the `step` function. Each call will take a single step in the iteration.<sup>1</sup>

### B. Computation Model

EvoX employs a *workflow* abstraction to perform computations and automatically invoke different instances (e.g., *Algorithm* and *Problem* modules) when they are ready to execute. In this section, we detail how EvoX translates a user-written program (Listing 2) into an automated workflow (Fig. 2) for parallel execution internally. At the core, an EC workflow is driven by executing an ask-evaluate-tell loop iteratively, which seamlessly integrates the *Algorithm*, *Problem*, and *Monitor* modules.

Initially, each iteration commences with the `ask` function in the *Algorithm* module to generate a population. To harness the capabilities of hardware acceleration, this population is encoded in a *tensor* format, i.e., tightly packed multidimensional arrays in memory for data-level parallelism. For simplicity, we denote the tensorized population data as  $\mathbf{T}_{pop} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_i, \dots]$  hereafter, where  $\mathbf{x}_i$  denotes each encoded candidate solution.

<sup>1</sup>Within the realm of EC algorithms, one iteration does not necessarily correspond to a single generation. Specifically, an iteration denotes a solitary ask-evaluate-tell loop, while some algorithms may require several iterations to complete a single generation.

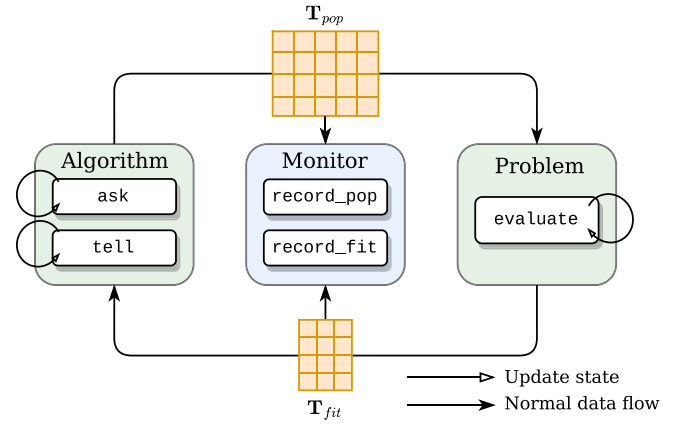


Fig. 2. Illustration of the computation model adopted by EvoX. There are three main modules: *Algorithm*, *Problem*, and *Monitor*. The iteration starts with `ask` function which generates  $\mathbf{T}_{pop}$  as the tensorized population. Then, the population is sent to the *Problem* module for fitness evaluations via `evaluate` function and then generate  $\mathbf{T}_{fit}$  as the tensorized fitness. Finally, the fitness is passed back to the *Algorithm* module through the `tell` function. Meanwhile,  $\mathbf{T}_{pop}$  and  $\mathbf{T}_{fit}$  can be optionally sent to the *Monitor* module for further processing, where `record_pop` and `record_fit` functions record  $\mathbf{T}_{pop}$  and  $\mathbf{T}_{fit}$ , respectively. In addition to the normal data flow, each module can update its individual state at every function call.

Upon generating the population, the `evaluate` function in the *Problem* module will return the fitness data, i.e., the corresponding fitness values of the candidate solutions within the population. Similar to  $\mathbf{T}_{pop}$ , the fitness data adopts a tensor representation to streamline subsequent processing. For simplicity, we denote the tensorized fitness data as  $\mathbf{T}_{fit} = [\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_i, \dots]$  hereafter, where  $\mathbf{y}_i$  denotes the fitness value of each candidate solution in the population.

Finally, the loop enters the `tell` function of the *Algorithm* module, which proceeds to select candidate solutions for generating new  $\mathbf{T}_{pop}$  according to  $\mathbf{T}_{fit}$ . Given the tensorial nature of both  $\mathbf{T}_{pop}$  and  $\mathbf{T}_{fit}$ , the *Algorithm* module can easily adapt to data-level parallelism.

Outside this primary loop, each module is responsible for two distinct data categories: 1) immutable data and 2) mutable data (i.e., the state). Immutable data, often initialized during object instantiation via Python's `__init__` method, encompass static hyperparameters pertinent to each module and might include external datasets for specific problems. These remain unchanged throughout the computational life cycle. In contrast, the state is initialized after the entire workflow has been assembled, starting with the topmost module and proceeding recursively through the dependency tree. During computation, as the global state is activated, state management ensures that each module receives its specific state data as the primary argument for its function. This segmented strategy enables each module to concentrate on updating its own state, while collaboratively participating in the comprehensive update of a unified global state.

One of the distinguishing attributes of EvoX is its foundational alignment with the tenets of functional programming. This approach accentuates a distinct separation between data and functions, thus ensuring that functions remain free from side effects. From this perspective, executing an EC algorithm

can be perceived as orchestrating state updates via a coherent data flow, which can be formally articulated as follows.

Given  $\theta$  as the hyperparameters, an EC algorithm  $\mathcal{A}_\theta$  can be denoted as  $\langle \theta, g^{\text{ask}}(\cdot), g^{\text{tell}}(\cdot) \rangle$ , where  $g^{\text{ask}}(\cdot)$  and  $g^{\text{tell}}(\cdot)$  correspond to the `ask` and `tell` functions, respectively. When  $\mathcal{A}_\theta$  is applied to a specific problem  $\mathcal{P}_\mathcal{D}$  with  $f^{\text{evl}}(\cdot)$  as its `evaluate` function,  $\mathcal{D}$  comprises the parameters or datasets related to the problem. At each step, the data flow is articulated as

$$\mathbf{T}_{\text{pop}}, S = g_\theta^{\text{ask}}(S) \quad (1)$$

$$\mathbf{T}_{\text{fit}}, S = f^{\text{evl}}(S, \mathbf{T}_{\text{pop}}) \quad (2)$$

$$S = g_\theta^{\text{tell}}(S, \mathbf{T}_{\text{fit}}) \quad (3)$$

where  $S$  denotes the state that encapsulates the mutable data associated with  $\mathcal{A}_\theta$  and  $\mathcal{P}_\mathcal{D}$ . From this formulation, it is evidenced that the only variable undergoing updates at each iterative step is the state  $S$ . Consequently, the operation of the entire EC algorithm can be envisioned as a sequence of state transitions, which is realized through the repeated execution of the `step` function within the *Workflow* module.

Beyond the primary computational processes of EC, the *Monitor* module also plays a pivotal role. Users are usually interested in the data generated during the running process of an EC algorithm, such as the best solution found so far, and the distribution of the population, among various other aspects. The *Monitor* module tailored this purpose in a *pluggable* and *asynchronous* manner. When the user plugs a monitor into the workflow, both  $\mathbf{T}_{\text{pop}}$  and  $\mathbf{T}_{\text{fit}}$  are asynchronously sent to the monitor. This allows the main EC computational process to progress uninterrupted, without waiting for the monitor to complete the data processing. Given that the monitor might involve subsequent slow disk I/O or plotting functions, this asynchronous design can significantly enhance the overall computational efficiency. Furthermore, since the monitor does not interfere with the primary computational process, the module even allows users to employ multiple monitors in parallel when needed.

To realize our envisioned workflow abstraction which coordinates these different modules, EvoX employs a *stateful computation* model for all primary modules. This model adopts the signature  $(\text{state}, \dots) \rightarrow (\text{result}, \text{state})$ , where each module operates based on its current state and other inputs, subsequently producing a result and an updated state. However, as we arrange these modules hierarchically, complexities arise: although it is naturally expected for a module to manage only its state, it also inherits the responsibility for the states of its nested submodules. To meet the requirement of such layered state management, EvoX adopts a *hierarchical state management* mechanism, ensuring the smooth crafting of stateful computations across intricate hierarchies. Central to this system is the *state* variable, which acts as a *universal conduit* across computational modules. Rather than being a static entity, this variable dynamically adjusts to encapsulate the relevant state during each function's execution. This adaptive design ensures automated state transitions and efficient management across the comprehensive hierarchical framework. In the following

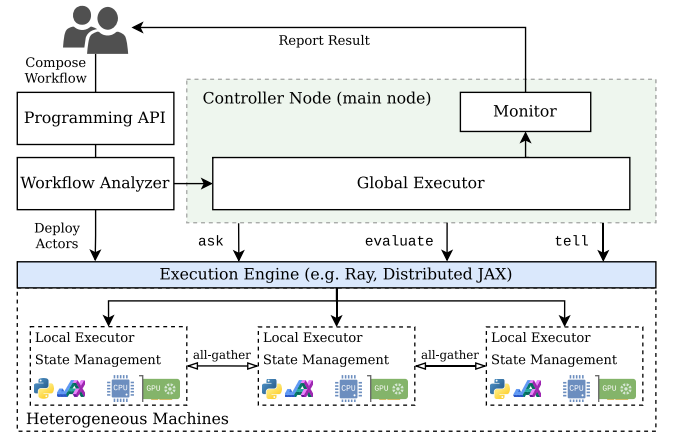


Fig. 3. Architecture of EvoX. The workflow analyzer sets the stage for task execution. Each node is equipped with a local workflow executor, responsible for orchestrating the ask-evaluate-tell loop. At the controller node, a global workflow executor directs the local workflow executors within this loop, employing the all-gather collective operation to harmonize fitness values obtained from the evaluate phase.

section, we will delve into the hierarchical state management mechanism, as well as other main components that underpin the architecture of EvoX.

## V. ARCHITECTURE

As illustrated by Fig. 3, the general architecture of EvoX comprises five main components: 1) programming API; 2) workflow analyzer; 3) global workflow running; 4) monitor; and 5) execution engine. Among them, the programming API and monitor have already been introduced in the previous section. In this section, we will first elaborate on the hierarchical state management mechanism, which is the core of the entire architecture. Then, we will delve into the other three main components: 1) workflow analyzer; 2) workflow executor; and 3) execution engine.

### A. Hierarchical State Management

As highlighted in the preceding section, EvoX utilizes a *hierarchical state management* mechanism to streamline state transitions throughout the execution process. To elucidate this mechanism, let us consider the hyperparameter tuning task illustrated in Fig. 4. The architecture is structured across three module tiers. At the pinnacle is a workflow encapsulating the entire hyperparameter optimization endeavor. This workflow branches into two distinct submodules: 1) an algorithm module (for optimizing the parameter set) and 2) a problem module (for assessing the performance of a specific parameter set). Initially, each module undergoes a recursive state initialization and is assigned a unique identifier. When a function accesses the state, it retrieves its own unique identifier and fetches the corresponding state, as depicted in Fig. 4(a). Post-execution, when the function returns an updated state, this state is seamlessly integrated back into the overarching universal state using the identifier, a process visualized in Fig. 4(b).

Although state management introduces additional overhead, it is effectively eliminated by leveraging the JAX's JIT

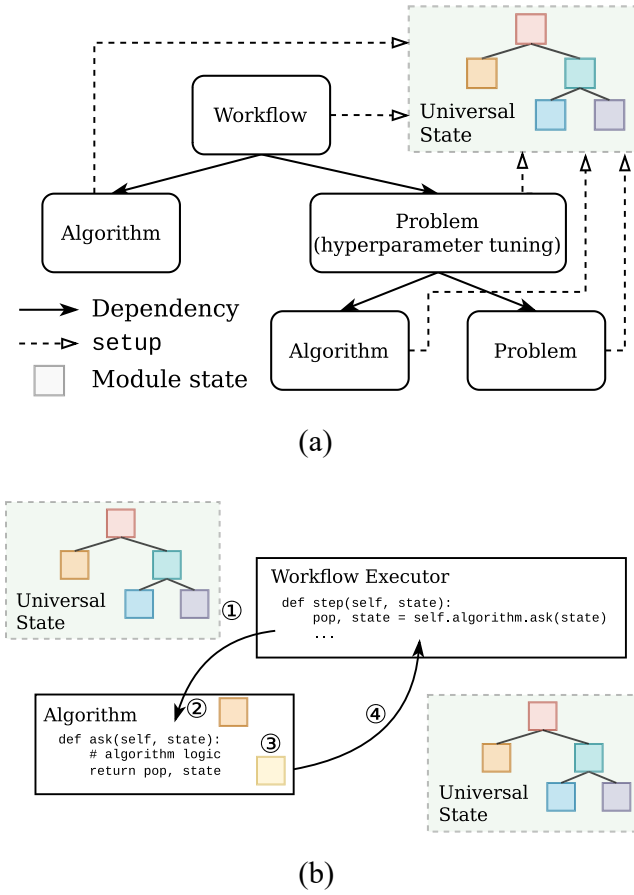


Fig. 4. Hierarchical state management in EvoX. (a) State initialization. (b) State update.

compiler. The system efficiently resolves the overhead associated with state identification after compilation. Specifically, JAX first traces the function, converting it into a computational graph where operations are directly linked to the tensors it applies to. Once this computational graph is compiled, subsequent function calls take advantage of this precompiled version, bypassing the need for repeated state identification and ensuring optimal performance.

Within the proposed hierarchical state management, traditional distinctions between algorithms and problems become nuanced: all elements present themselves as interconnected modules within a comprehensive hierarchical architecture. This design aligns with the requirement of *Flexibility and Extensibility* as discussed in Section III.

### B. Workflow Analyzer

As depicted in Fig. 5, the workflow analyzer is instrumental in analyzing and tailoring the execution behavior to align with user requirements. Given EvoX's commitment to the functional programming paradigm, workflows can be modularly decomposed.

In multidevice scenarios, despite the presence of a single host, a multitude of diverse devices can be concurrently harnessed. When leveraging multiple local devices, strategic data distribution becomes paramount. Primarily, two strategies

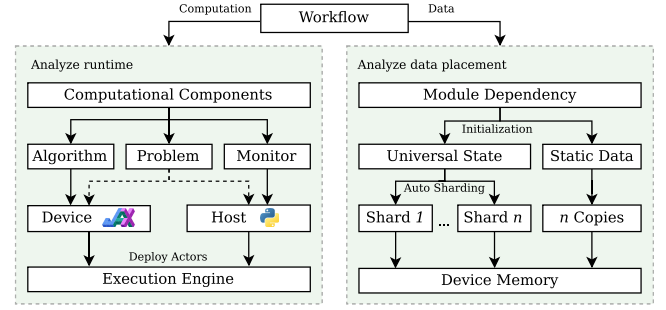


Fig. 5. Workflow analyzer in EvoX. The analyzer examines both computation and data components distinctly. For computation, it ascertains the optimal runtime: typically, algorithms execute on devices, monitors operate on the host, and the problem can run on either. For data, it strategically distributes data across multiple devices. The universal state is sharded to ensure distribution across all devices, while the static data is replicated consistently to each device.

emerge: 1) *replication*—duplicating the same data across all devices and 2) *sharding*—each device holding only a portion of the complete dataset. By aggregating the capabilities of multiple devices, sharding optimizes individual device memory and can handle larger datasets than a standalone device. However, it may necessitate data transfer between devices for certain operations, thus introducing potential communication overheads. This underscores the need for sharding strategies that mitigate communication costs.

From a data flow perspective, the crux of EC lies in the repetitive update of the fitness and population tensors,  $T_{fit}$  and  $T_{pop}$ , as delineated in Section IV-B. Given this, it is imperative that any sharding strategy should consider both tensors. For  $T_{fit}$ , it is straightforward that the evaluation of each candidate solution can be performed independently and in parallel. However, for  $T_{pop}$ , which represents the tensor of encoded candidate solutions, the situation is more sophisticated. Basically, there exist two potential sharding strategies: segmenting by individual encoded candidates or by encoded dimensions. While the former strategy seems intuitive, it is less effective in the context of EC algorithms, which may frequently require collective information from the entire population for estimation of distribution. In contrast, sharding by encoded dimension aligns better with the inherent behavior of EC algorithms. During reproductive operations such as crossover or mutation, EC algorithms typically adjust each dimension of the individual candidate solutions in an isolated manner. Hence, EvoX adopts *dimension-centric sharding* for  $T_{pop}$  to curtail cross-device communication overheads.

It is important to note that while the sharding strategy can be applied universally across various algorithms, its effectiveness can be influenced by specific algorithm designs. Particularly, algorithms that engage extensively in cross-dimensional operations may not fully benefit the multidevice acceleration.

In distributed settings, the workflow analyzer synchronizes both the module and its preliminary data across nodes. When a sophisticated distributed engine like Ray is employed, the analyzer wraps both the algorithm and problem, including their states, into actor constructs. With the execution engine's aid, these actors are dispatched to each node. However, when using

a basic engine like distributed JAX, the workflow analyzer employs the single program–multiple data (SPMD) pattern. Here, users are tasked with initiating an identical program on all nodes, but with slight input variations (e.g., node IDs). This approach ensures natural synchronization during the initialization phase, resulting in uniform module and data distribution across nodes.

Moreover, our workflow analyzer is capable of effectively separating JAX code from standard Python code. Upon isolation, the JAX code undergoes compilation by XLA, thus facilitating its application across a diverse array of hardware backends, such as CPUs, NVIDIA GPUs, and AMD GPUs, among others. This adaptability offers users the flexibility to effortlessly transition between execution backends to meet their unique requirements.

### C. Workflow Executor

In multidevice configurations, even after the state partitioning facilitated by the workflow analyzer, the sharding of all intermediate tensors remains paramount. To this end, we adopt the GSPMD method [22] as embraced by JAX, which ensures consistent sharding propagation for every intermediate tensor. Within the realm of distributed computing, our approach focuses on maximizing multidevice capabilities within each node. In contrast to specific distributed EC algorithms (e.g., the distributed ES [23]), our approach embeds itself at the workflow level. This architectural choice endows EvoX with unparalleled adaptability, which enables support for a myriad of algorithms and problems without mandating foundational code alterations.

In detail, the workflow executor disperses the *Algorithm* and *Problem* modules as actors across each node while concurrently mirroring the state. On the one hand, every node employs a local workflow executor tailored for the ask-evaluate-tell loop. On the other hand, a centralized global workflow executor oversees the entire node network's execution for coordinating operations across local executors. Central to its approach is the all-gather collective operation which synchronizes fitness values returned by the `evaluate` function. This synchronized aggregation paves the way for a unified update during the `tell` step, thus ensuring end-to-end synchronization at the end of each iteration.

### D. Execution Engine

The execution engine refers to the component specially engineered to coordinate and synchronize the execution of code across disparate machines. Within the architecture of EvoX, we predominantly leverage two salient execution engines: 1) Ray and 2) distributed JAX. As a Python-centric framework, Ray is underpinned by an actor-based programming paradigm. Users can delegate actors to Ray's scheduler, which then automatically allocates these actors to machines in accordance with stipulated computational requirements. In contrast, the distributed functionality within JAX deviates from that of Ray. Rather than incorporating scheduling capabilities,

TABLE II  
SELECTED EC ALGORITHMS IN EVOX FOR SINGLE-OBJECTIVE OPTIMIZATION: EVOLUTION STRATEGY (ES), PARTICLE SWARM OPTIMIZATION (PSO), AND DIFFERENTIAL EVOLUTION (DE)

Type	Algorithm Name
ES	CMA-ES [24], PGPE [25], OpenES [23], CR-FM-NES [26], xNES [27], ...
PSO	FIPS [28], CSO [29], CPSO [30], CLPSO [31], SL-PSO [32], ...
DE	CoDE [33], JaDE [34], SaDE [35], SHADE [36], IMODE [37], ...

TABLE III  
SELECTED EC ALGORITHMS IN EVOX FOR MULTIOBJECTIVE OPTIMIZATION: DOMINANCE-BASED, DECOMPOSITION-BASED, AND INDICATOR-BASED APPROACHES

Type	Algorithm Name
Dominance-based	NSGA-II [38], NSGA-III [39], SPEA2 [40], BiGE [41], KnEA [42], ...
Decomposition-based	MOEA/D [43], RVEA [44], t-DEA [45], MOEAD-M2M [46], EAG-MOEAD [47], ...
Indicator-based	IBEA [48], HypE [49], SRA [50], MaOEA-IGD [51], AR-MOEA [52], ...

this feature of JAX prioritizes synchronization tools and facilitates collective operations across a multitude of nodes.

## VI. IMPLEMENTATION

EvoX is developed in Python and leverages JAX for optimized execution on hardware accelerators. The design integrates distributed computing features using either Ray or distributed JAX. For the implementation using Ray, modules and their respective states are treated as actors, and assigned to Ray's scheduler. For the implementation using distributed JAX, users are asked to initiate a consistent program across nodes, but with distinctions in input arguments. Based on this foundation, we have further crafted a comprehensive library. For user convenience, EvoX is readily available on PyPI, facilitating an effortless installation process via the `pip` command. Moreover, we provide a comprehensive documentation<sup>2</sup> for detailed user guidance.

In detail, for single-objective optimization, EvoX presents EC algorithms spanning categories, such as ES, particle swarm optimization (PSO), and differential evolution (DE), as listed in Table II. for multiobjective optimization, the library covers all three representative types of EC algorithms: dominance-based, decomposition-based, and indicator-based ones, as listed in Table III. Beyond algorithms, EvoX also provides seamless support for a diverse set of benchmark problems, which fulfills various research and application requirements. This collection includes benchmark test suites for numerical optimization and environments for reinforcement learning tasks, as listed in Table IV. All algorithms and problems are implemented in a tensorized manner to maximize the parallelization benefits of GPU acceleration.

<sup>2</sup>The documentation of EvoX is available at: <https://evox.readthedocs.io/>.



TABLE IV  
BENCHMARK PROBLEMS PROVIDED BY EvoX

Name	Description
CEC'22 [53]	A test suite for benchmarking single-objective numerical optimization.
ZDT [54]	A test suite for benchmarking bi-objective numerical optimization.
DTLZ [55]	A test suite for benchmarking multi-objective numerical optimization with scalable dimensions.
MaF [56]	A test suite for benchmarking multi-objective numerical optimization with many objectives, a.k.a., many-objective optimization.
LSMOP [57]	A test suite for benchmarking multi-objective numerical optimization with large-scale decision variables.
Brax [58]	A JAX-based physics engine, fully differentiable and optimized for reinforcement learning, robotics, and other simulation-intensive applications.
Gym [59]	A comprehensive toolkit for developing and comparing reinforcement learning algorithms through a collection of standardized environments.

## VII. EXPERIMENTAL STUDY

This section offers a comprehensive assessment of the performance of EvoX through a series of experiments. Except for the multinode acceleration experiment presented in Section VII-A3, all the other experiments were executed on a dedicated machine equipped with two Intel Xeon Gold 6226R CPU @ 2.90 GHz processors (each with 16 cores and 32 threads)<sup>3</sup> and eight NVIDIA A100 GPUs, with only one GPU engaged for each experiment. In contrast, the experiment in Section VII-A3 utilized a 4-node physical GPU cluster interconnected using 10-Gb Ethernet over optical fiber, where each node integrated an Intel Xeon Gold 6240 CPU @ 2.60 GHz and four NVIDIA RTX 2080ti GPUs. The pseudo-random number generation remained consistent, employing the `rbg` generator from JAX.

### A. System Performance

The primary objective of this section is to evaluate the efficacy of GPU acceleration in handling both single-objective and multiobjective numerical optimization tasks. Following this, we scrutinize the framework's proficiency in multinode acceleration.

1) *Single-Objective Numerical Optimization*: In this experiment, we evaluate three representative EC algorithms (PSO [60], DE [61], and CMA-ES [24]) on the Sphere function, which is a basic benchmark function for single-objective numerical optimization. During the assessment of scaling with respect to the problem dimension, we retained a consistent population size of 100. On the other hand, in the evaluation of population size scalability, we maintained the problem dimension at a constant value of 100.

Fig. 6 reveals that GPU acceleration considerably enhances the performance of the evaluated algorithms, especially as the problem's dimension or population size grows. Initial

<sup>3</sup>For experiments running on CPUs, parallelization was optimized across all  $2 \times 32$  threads.

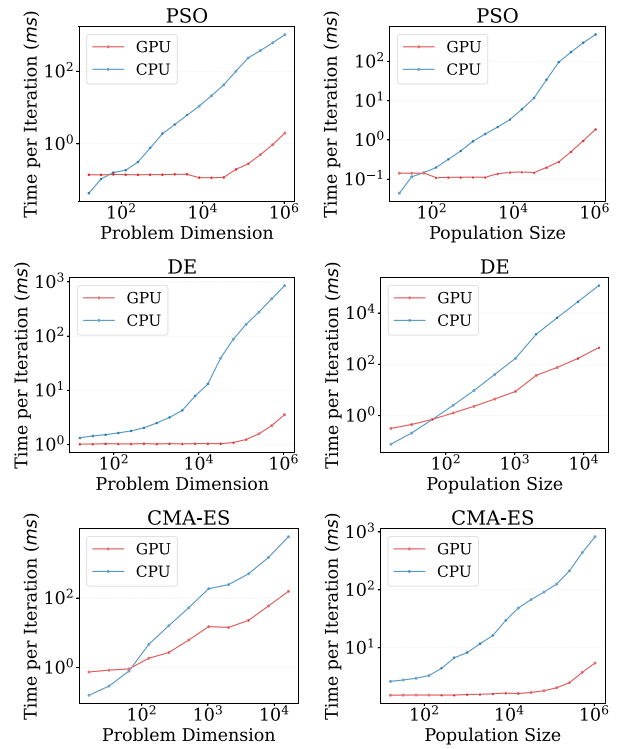


Fig. 6. Scalability performance of PSO, DE, and CMA-ES on the Sphere function for single-objective numerical optimization, in terms of problem dimension and population size. Both axes employ a logarithmic scale. A fixed population size of 100 is used when scaling the problem dimension, and vice versa.

tests with small dimensions or populations might favor the CPU, but GPU performance rapidly overtakes as the scale increases, frequently achieving a tenfold or greater speedup. A notable observation is the plateauing of performance in the early stages of the scaling tests with GPU acceleration. This plateau suggests that lighter computations cannot fully harness the GPU's capabilities, thus resulting in near-constant computational costs.

However, it is paramount to understand that the advantages of GPU acceleration are algorithm-dependent. Algorithms inherently unsuited for parallelism or those restricted by memory constraints might not benefit as significantly. For instance, CMA-ES internally uses a covariance matrix, demanding memory proportional to the square of the problem dimension. This requirement limited its ability to scale beyond a dimension of 16384, even though the GPU accelerated performance by orders of magnitude. The vanilla DE also presents some limitations, particularly when scaling the population size beyond 16384. Such restrictions emerge from specific operators within DE not optimized for GPUs or larger populations. Specifically, the mutation operator in vanilla DE, while suitable for smaller populations, becomes computationally intensive with larger populations, especially when ensuring distinct individual sampling.

2) *Multiobjective Numerical Optimization*: This section evaluates the advantages of GPU acceleration for multiobjective EC algorithms, segmented into two primary investigative parts. First, we investigated the scalability of

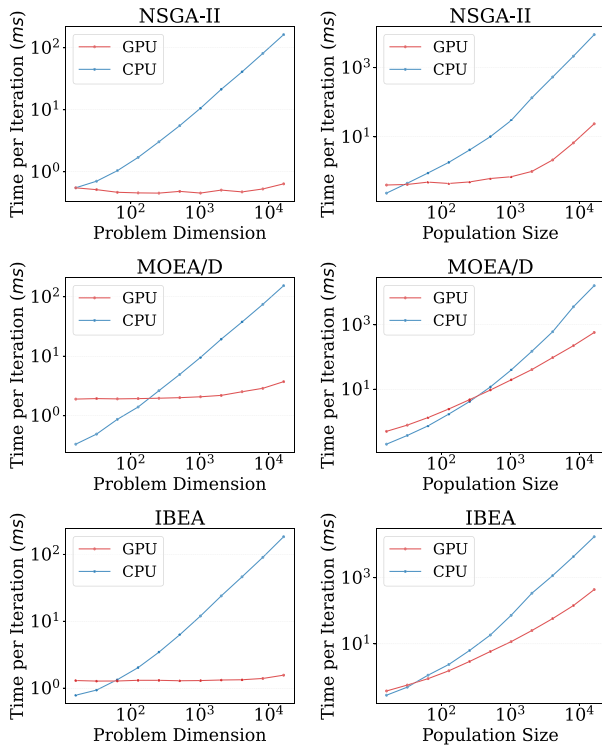


Fig. 7. Scalability performance of NSGA-II, MOEA/D, and IBEA on the DTLZ1 function for multiobjective numerical optimization, in terms of problem dimension and population size. Both axes employ a logarithmic scale. A fixed population size of 100 is used when scaling the problem dimension, and vice versa.

three representative multiobjective EC algorithms: 1) NSGA-II [38]; 2) MOEA/D [43]; and 3) IBEA [48]. When scaling the problem dimension, we consistently employed a population size of 100, and vice versa. Subsequently, we assessed the scalability of another three representative multiobjective EC algorithms: NSGA-III [39], RVEA[44], and HypE [49] in relation to the number of optimization objectives. For this part, the problem dimension was fixed at 100 000. All experiments were conducted using DTLZ1 [55], one of the most commonly used test functions for multiobjective numerical optimization.

As shown in Fig. 7, NSGA-II notably benefits from GPU acceleration during both problem dimension and population size scaling. While MOEA/D's scalability is not as pronounced as NSGA-II, it still substantially benefits from GPU acceleration. IBEA also demonstrates significant speed enhancements, particularly when the problem dimension is high.

Notably, the effectiveness of GPU acceleration is intrinsically related to the algorithmic mechanisms. For example, the performance improvement in NSGA-II is attributed to the GPU-accelerated computation of dominance relations, which is computationally intensive but amenable to parallelization. In contrast, the scalability of MOEA/D is somehow limited by its inherent design. Specifically, the sequential update strategy within the reproduction operator requires the completion of one individual's update before proceeding to the next. This sequential dependency hampers the potential for parallel processing, which is crucial for GPU acceleration.

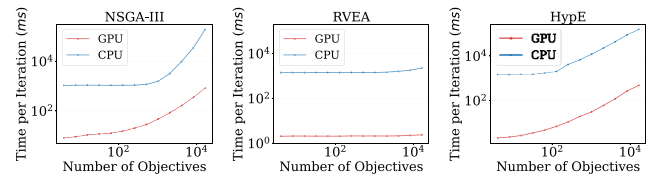


Fig. 8. Scalability performance of NSGA-III, RVEA, and HypE on the DTLZ1 function for multiobjective numerical optimization, in terms of the number of optimization objectives. Both axes employ a logarithmic scale. The problem dimension is fixed at 100 000.

TABLE V  
ARCHITECTURE OF THE CNN USED IN THE MULTINODE  
ACCELERATION EXPERIMENT

Input Shape	Layer	Filter Shape	Strides
$32 \times 32 \times 3$	Conv	$3 \times 3 \times 3 \times 32$	1
$30 \times 30 \times 32$	Max Pooling	$2 \times 2$	2
$15 \times 15 \times 32$	Conv	$3 \times 3 \times 32 \times 32$	1
$13 \times 13 \times 32$	Max Pooling	$2 \times 2$	2
$6 \times 6 \times 32$	Conv	$3 \times 3 \times 32 \times 32$	1
512	Fully Connected	$512 \times 64$	—
64	Fully Connected	$64 \times 10$	—

Fig. 8 indicates that all tested algorithms significantly benefit from GPU acceleration as the number of optimization objectives increases. Although these algorithms were not inherently designed for a large number of optimization objectives, they maintain consistent performance up to 100 objectives. However, as the count escalates, NSGA-III and HypE's performance diminishes while RVEA remains consistently robust, although the scenarios involving over 100 optimization objectives are rare in practice.

3) *Multinode Acceleration*: This section evaluates the efficacy of multinode acceleration and contrasts the performance between the two execution engines leveraging JAX and Ray, respectively. For this purpose, we conducted an experiment on neuroevolution for image classification across multiple GPU devices.

Specifically, we evolved a convolutional neural network (CNN) on the CIFAR-10 dataset [62], scaling from 4 to 16 GPUs, and measured the time per iteration. The CNN architecture, as detailed in Table V, employs ReLU [63] as the activation function between layers. For the EC algorithm, we utilized PGPE [25] with a population size of 192. To quantify the acceleration's effectiveness, we present two metrics: time per iteration and relative performance, the latter being the inverse of the former.

Fig. 9 presents the performance of multinode acceleration. Notably, for a lower count of GPU nodes, the runtime substantially reduces with increased GPU nodes, achieving an almost linear acceleration rate. However, as more GPUs are integrated, the rate of performance gain tapers off, leading to an overarching sublinear acceleration trend. This behavior aligns with expectations. While our acceleration framework primarily accelerates the computational parts of the workflow, the distributed execution engine introduces some overhead. As GPU count escalates, the cost of fitness evaluations drops, thus making other workflow costs more prominent.

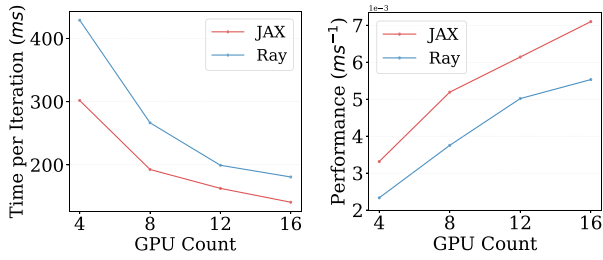


Fig. 9. Results of multinode acceleration using 4–16 GPUs. Data is presented in two forms: runtime (left) and its inverse (right).

Notably, the scalability of our distributed workflow is closely tied to the problem’s nature. More computationally demanding problems offer better scalability since the distributed framework effectively offloads the fitness evaluations, thus yielding substantial gains. In contrast, for computationally cheap problems (e.g., numerical optimization), the performance enhancements can be less significant. This is because the algorithm’s demands and the distributed framework’s overhead often outweigh the computational cost of fitness evaluations.

Besides, the two execution engines leveraging JAX and Ray possess unique performance attributes, respectively. JAX incurs relatively lesser overhead and offers superior scalability, significantly outperforming Ray on a 4-GPU setup. This performance disparity stems from JAX’s efficient hardware utilization, especially when recognizing that the 4 GPUs reside on the same physical node. In contrast, Ray offers a more intuitive interface, supplemented by features like scheduling and fault tolerance, which are capabilities absent in distributed JAX. Ray’s scheduling allows users to initiate a task once, distributing it automatically across nodes, while JAX mandates manual task initiation on each node.

### B. Model Performance

Within EvoX, we have seamlessly integrated a range of black-box optimization challenges into the *Problem* module, all adhering to a unified interface. Among these, the reinforcement learning tasks stand out as particularly intricate. To assess the model performance of EvoX, we present two distinct demonstrations: one leveraging the CPU-centric Gym [59] platform, and the other utilizing the GPU-accelerated Brax [58] platform. In both cases, the *Problem* module proficiently manages the interaction between the policy models and the reinforcement learning environments, enabling the EC algorithm to singularly concentrate on refining the policy model’s weights through neuroevolution, independent of the specificities of the task at hand. For a comprehensive evaluation of EvoX’s capabilities, we benchmarked an ES algorithm (PGPE [25]), in comparison with the widely acknowledged baseline (PPO2 baseline [64]) as endorsed by OpenAI [65].

1) *Performance With Gym*: Over the years, Gym has emerged as an essential open-source platform for developing and benchmarking reinforcement learning algorithms. It offers a plethora of predefined environments, streamlining the testing

```
1 problem = Gym(
2     env_name=..., # Gym's environment name
3     policy=..., # your policy
4     num_workers=..., # number of CPU workers
5     env_per_worker=..., # environments per worker
6 )
```

Listing 3. Configuration for setting up a Gym-based reinforcement learning task in EvoX. Users simply need to specify the environment, define the policy model, and adjust runtime parameters to optimize CPU utilization.

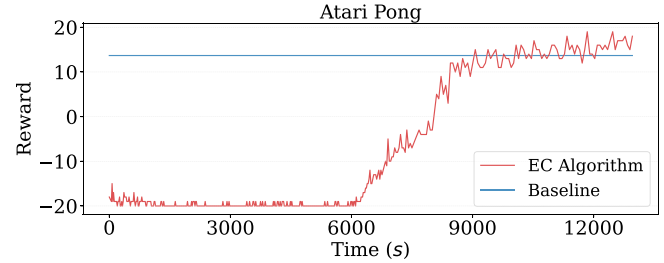


Fig. 10. Performance curve when tackling the Atari Pong task with Gym. The EC algorithm is PGPE with a population size of 256. The baseline performance, achieved by PPO2, was directly taken from [65] as a reference.

```
1 problem = Brax(
2     env_name=..., # Brax's environment name
3     policy=..., # user's policy model
4     batch_size=..., # concurrency of environments
5 )
```

Listing 4. Configuration for setting up a Brax-based reinforcement learning task in EvoX. Users simply need to specify the environment, define the policy model, and determine the *batch\_size* for specifying the number of environments running in concurrency.

and comparison of various algorithms on a standardized platform. In 2021, the development of Gym transitioned to Gymnasium [66], serving as a direct replacement.

As illustrated in Listing 4, setting up a Gym-based problem in EvoX is straightforward. Users simply specify a Gym-supported environment and define the policy network’s forward function. This function primarily accepts two inputs: the network’s weight and the observational data from the environment. The EC algorithm outputs a varied set of weights for the policy network, which are then evaluated within the specified Gym environment to aggregate rewards (i.e., fitness values). Additionally, the runtime configuration can be tailored to best align with users’ computational resources and needs.

Specifically, we instantiated an Atari Pong task with the policy model being a CNN with 78 102 parameters. As shown in Fig. 10, the complexity of Atari Pong and the constraints of the CPU-centric game emulator significantly affected the speed of the workflow. Nonetheless, thanks to EvoX’s efficient architecture, all available CPU cores were maximized, accomplishing the tasks in roughly 4 h to reach the baseline performance.

2) *Performance With Brax*: Brax is a differentiable physics engine developed in JAX, which capitalizes on JAX’s capabilities to harness GPUs for extensive parallel simulations. Given that EvoX shares its foundation with JAX, it seamlessly integrates with Brax.

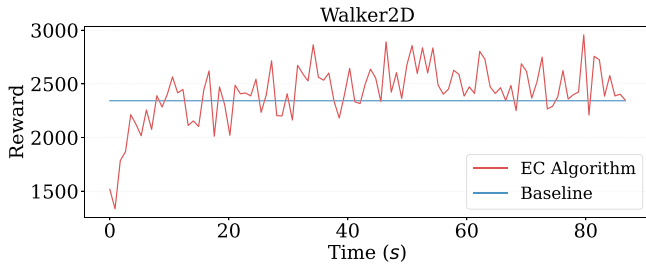


Fig. 11. Performance curve when tackling the Hopper task with Brax. The EC algorithm employed is CMA-ES with a population size of 4096. The baseline performance, achieved by PPO2, was directly taken from [65] as a reference.

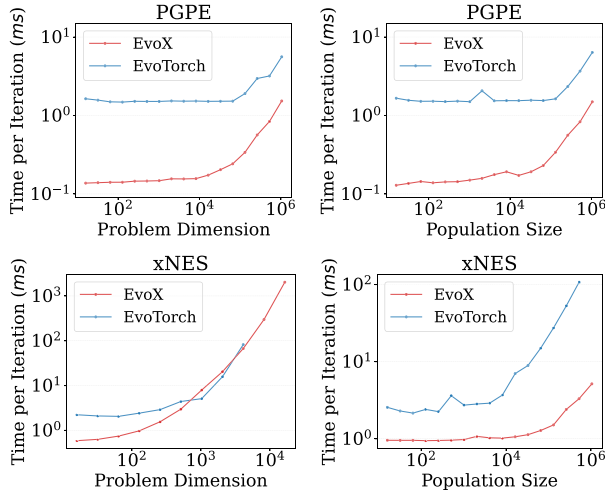


Fig. 12. Comparative performance with EvoTorch using the Sphere function for PGPE and xNES. Both axes are on a logarithmic scale. For tests where the problem dimension is scaled, the population size is fixed at 100, and vice versa. Note: EvoTorch's tests of xNES were terminated due to an *Out-of-Memory* error.

As illustrated in Listing 3, setting up a Brax-centric problem in EvoX is similar to the case with Gym, necessitating the environment's name and the forward function for the policy network. A unique aspect of Brax is the `batch_size` parameter, which indicates the number of concurrent environments on the hardware accelerator. This often aligns with the population size of an EC algorithm to harness the prowess of Brax by batch-evaluating the environments on GPU(s).

Specifically, we instantiated a Walker2D task with the policy model being a 3-layer MLP with 1830 parameters. As shown in Fig. 11, with GPU acceleration, the EC algorithm was able to achieve the baseline performance within approximately 1 min, underscoring the promising potential of EvoX in tackling reinforcement learning tasks via neuroevolution.

### C. Comparison With EvoTorch

To further benchmark the efficiency of EvoX, we juxtaposed it against EvoTorch [17], a library built atop PyTorch. We concentrated on evaluating two natively supported algorithms by both EvoX and EvoTorch: PGPE and xNES.

As evidenced in Fig. 12, EvoX exhibits promising performance in comparison to EvoTorch. PGPE's execution on EvoX is generally faster across various configurations. In

the case of xNES, EvoX also tends to be more efficient, especially during population scaling tests. When the dimension increased to 8192, EvoTorch encountered an *Out-of-Memory* error and thus failed to continue the test. A similar issue arose when the xNES population size was increased to 1 048 576. In contrast, EvoX managed to complete the benchmark on the same hardware setup, indicating its efficiency in memory management.

## VIII. CONCLUSION

Throughout its history, EC has proven to be an effective tool in addressing a wide array of problems across numerous domains. However, the rise of large-scale data and complex systems has posed significant scalability challenges for it. In response, we have developed EvoX, a computing framework specifically crafted for scalable EC. This framework, with its tailored programming and computation models coupled with a hierarchical state management system, facilitates the efficient employment of distributed and heterogeneous computational resources.

Designed with flexibility and extensibility in mind, EvoX is committed to ongoing development toward various application areas. Among the promising directions for future development are evolutionary multitasking [67] and evolutionary transfer optimization [68]. These areas, characterized by computationally intensive yet inherently parallelizable workloads, can significantly benefit from GPU acceleration. Furthermore, as advancements in computing architectures continue to reshape the technological landscape, EvoX is poised for ongoing refinement. This will guarantee the sustained relevance of EC within the rapidly changing domain of AI.

## ACKNOWLEDGMENT

The authors thank Minyang Chen, Jiachun Li, Zhenyu Liang, Kebin Sun, Lishuang Wang, Haoming Zhang, and Mengfei Zhao for their efforts in helping with the implementations and testings. In particular, they thank Yansong Huang for his unique contributions to addressing the visualization issues related to the entire project.

## REFERENCES

- [1] T. Bäck, D. B. Fogel, and Z. Michalewicz, "Handbook of Evolutionary Computation, vol. 97. Bristol, U.K.: IOP Publ., 1997.
- [2] C. A. Pena-Reyes and M. Sipper, "Evolutionary computation in medicine: An overview," *Artif. Intell. Med.*, vol. 19, no. 1, pp. 1–23, 2000.
- [3] I. C. Parmee, *Evolutionary and Adaptive Computing in Engineering Design*. London, U.K.: Springer, 2012.
- [4] H. Malik, A. Iqbal, P. Joshi, S. Agrawal, and F. I. Bakhsh, *Metaheuristic and Evolutionary Computation: Algorithms and Applications*. Singapore: Springer, 2021, vol. 916.
- [5] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkilainen, "Designing neural networks through neuroevolution," *Nat. Mach. Intell.*, vol. 1, no. 1, pp. 24–35, 2019.
- [6] M. N. Omidvar, X. Li, and X. Yao, "A review of population-based metaheuristics for large-scale black-box global optimization—Part II," *IEEE Trans. Evol. Comput.*, vol. 26, no. 5, pp. 823–843, Oct. 2022.
- [7] S. Liu, Q. Lin, J. Li, and K. C. Tan, "A survey on learnable evolutionary algorithms for scalable multiobjective optimization," *IEEE Trans. Evol. Comput.*, vol. 27, no. 6, pp. 1941–1961, Dec. 2023.



- [8] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan, "A survey on evolutionary neural architecture search," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 2, pp. 550–570, Feb. 2023.
- [9] Z.-H. Zhan, J.-Y. Li, and J. Zhang, "Evolutionary deep learning: A survey," *Neurocomputing*, vol. 483, pp. 42–58, Apr. 2022.
- [10] R. Miikkulainen and S. Forrest, "A biological perspective on evolutionary computation," *Nat. Mach. Intell.*, vol. 3, no. 1, pp. 9–15, 2021.
- [11] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *J. Mach. Learn. Res.*, vol. 13, pp. 2171–2175, Jul. 2012.
- [12] A. F. Gad, "PyGAD: An intuitive genetic algorithm python library," *Multimedia Tools Appl.*, pp. 1–4, Dec. 2023. [Online]. Available: <https://link.springer.com/article/10.1007/s11042-023-17167-y>
- [13] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89497–89509, 2020.
- [14] F. Biscani and D. Izzo, "A parallel global multiobjective framework for optimization: Pagmo," *J. Open Source Softw.*, vol. 5, no. 53, p. 2338, 2020.
- [15] Y. Tang, Y. Tian, and D. Ha, "EvoJAX: Hardware-accelerated neuroevolution," 2022, *arXiv:2202.05008*.
- [16] R. T. Lange, "evosax: JAX-based evolution strategies," 2022, *arXiv:2212.04180*.
- [17] N. E. Toklu, T. Atkinson, V. Micka, P. Liskowski, and R. K. Srivastava, "EvoTorch: Scalable evolutionary computation in python," 2023, *arXiv:2302.12600*.
- [18] J. Bradbury et al., "JAX: Composable transformations of python+NumPy programs," 2018. [Online]. Available: <http://github.com/google/jax>
- [19] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.
- [20] P. Moritz et al., "Ray: A distributed framework for emerging AI applications," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 561–577.
- [21] H. Bai, R. Cheng, and Y. Jin, "Evolutionary reinforcement learning: A survey," *Intell. Comput.*, vol. 2, p. 25, May 2023.
- [22] Y. Xu et al., "GSPMD: General and scalable parallelization for ML computation graphs," 2021, *arXiv:2105.04663*.
- [23] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," 2017, *arXiv:1703.03864*.
- [24] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)," *Evol. Comput.*, vol. 11, no. 1, pp. 1–18, Mar. 2003.
- [25] F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber, "Parameter-exploring policy gradients," *Neural Netw.*, vol. 23, no. 4, pp. 551–559, 2010.
- [26] M. Nomura and I. Ono, "Fast moving natural evolution strategy for high-dimensional problems," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, 2022, pp. 1–8.
- [27] T. Glasmachers, T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber, "Exponential natural evolution strategies," in *Proc. 12th Annu. Conf. Genetic Evol. Comput.*, 2010, pp. 393–400.
- [28] R. Mendes, J. Kennedy, and J. Neves, "The fully informed particle swarm: Simpler, maybe better," *IEEE Trans. Evol. Comput.*, vol. 8, no. 3, pp. 204–210, Jun. 2004.
- [29] R. Cheng and Y. Jin, "A competitive swarm optimizer for large scale optimization," *IEEE Trans. Cybern.*, vol. 45, no. 2, pp. 191–204, Feb. 2015.
- [30] F. van den Bergh and A. Engelbrecht, "A cooperative approach to particle swarm optimization," *IEEE Trans. Evol. Comput.*, vol. 8, no. 3, pp. 225–239, Jun. 2004.
- [31] J. Liang, A. Qin, P. Suganthan, and S. Baskar, "Comprehensive learning particle swarm optimizer for global optimization of multimodal functions," *IEEE Trans. Evol. Comput.*, vol. 10, no. 3, pp. 281–295, Jun. 2006.
- [32] R. Cheng and Y. Jin, "Demonstrator selection in a social learning particle swarm optimizer," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, 2014, pp. 3103–3110.
- [33] Y. Wang, Z. Cai, and Q. Zhang, "Differential evolution with composite trial vector generation strategies and control parameters," *IEEE Trans. Evol. Comput.*, vol. 15, no. 1, pp. 55–66, Feb. 2011.
- [34] J. Zhang and A. C. Sanderson, "JADE: Adaptive differential evolution with optional external archive," *IEEE Trans. Evol. Comput.*, vol. 13, no. 5, pp. 945–958, Oct. 2009.
- [35] A. K. Qin, V. L. Huang, and P. N. Suganthan, "Differential evolution algorithm with strategy adaptation for global numerical optimization," *IEEE Trans. Evol. Comput.*, vol. 13, no. 2, pp. 398–417, Apr. 2009.
- [36] R. Tanabe and A. Fukunaga, "Success-history based parameter adaptation for differential evolution," in *Proc. IEEE Congr. Evol. Comput.*, 2013, pp. 71–78.
- [37] K. M. Sallam, S. M. Elsayed, R. K. Chakraborty, and M. J. Ryan, "Improved multi-operator differential evolution algorithm for solving unconstrained problems," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, 2020, pp. 1–8.
- [38] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.
- [39] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints," *IEEE Trans. Evol. Comput.*, vol. 18, no. 4, pp. 577–601, Aug. 2014.
- [40] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength Pareto evolutionary algorithm," *Comput. Eng. Netw. Lab., ETH Zurich, Zurich, Switzerland, Rep. TIK-103*, 2001.
- [41] M. Li, S. Yang, and X. Liu, "Bi-goal evolution for many-objective optimization problems," *Artif. Intell.*, vol. 228, pp. 45–65, Nov. 2015.
- [42] X. Zhang, Y. Tian, and Y. Jin, "A knee point-driven evolutionary algorithm for many-objective optimization," *IEEE Trans. Evol. Comput.*, vol. 19, no. 6, pp. 761–776, Dec. 2015.
- [43] Q. Zhang and H. Li, "MOEA/D: A multiobjective evolutionary algorithm based on decomposition," *IEEE Trans. Evol. Comput.*, vol. 11, no. 6, pp. 712–731, Dec. 2007.
- [44] R. Cheng, Y. Jin, M. Olhofer, and B. Sendhoff, "A reference vector guided evolutionary algorithm for many-objective optimization," *IEEE Trans. Evol. Comput.*, vol. 20, no. 5, pp. 773–791, Oct. 2016.
- [45] Y. Yuan, H. Xu, B. Wang, and X. Yao, "A new dominance relation-based evolutionary algorithm for many-objective optimization," *IEEE Trans. Evol. Comput.*, vol. 20, no. 1, pp. 16–37, Feb. 2016.
- [46] H.-L. Liu, F. Gu, and Q. Zhang, "Decomposition of a multiobjective optimization problem into a number of simple multiobjective subproblems," *IEEE Trans. Evol. Comput.*, vol. 18, no. 3, pp. 450–455, Jun. 2014.
- [47] X. Cai, Y. Li, Z. Fan, and Q. Zhang, "An external archive guided multiobjective evolutionary algorithm based on decomposition for combinatorial optimization," *IEEE Trans. Evol. Comput.*, vol. 19, no. 4, pp. 508–523, Aug. 2015.
- [48] E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *Proc. Int. Conf. Parallel Probl. Solving Nat.*, 2004, pp. 832–842.
- [49] J. Bader and E. Zitzler, "HypE: An algorithm for fast hypervolume-based many-objective optimization," *Evol. Comput.*, vol. 19, no. 1, pp. 45–76, Mar. 2011.
- [50] B. Li, K. Tang, J. Li, and X. Yao, "Stochastic ranking algorithm for many-objective optimization based on multiple indicators," *IEEE Trans. Evol. Comput.*, vol. 20, no. 6, pp. 924–938, Dec. 2016.
- [51] Y. Sun, G. G. Yen, and Z. Yi, "IGD indicator-based evolutionary algorithm for many-objective optimization problems," *IEEE Trans. Evol. Comput.*, vol. 23, no. 2, pp. 173–187, Apr. 2019.
- [52] Y. Tian, R. Cheng, X. Zhang, F. Cheng, and Y. Jin, "An indicator-based multiobjective evolutionary algorithm with reference point adaptation for better versatility," *IEEE Trans. Evol. Comput.*, vol. 22, no. 4, pp. 609–622, Aug. 2018.
- [53] A. Ahrari, S. Elsayed, R. Sarker, D. Essam, and C. A. C. Coello, "Problem definition and evaluation criteria for the CEC'2022 competition on dynamic multimodal optimization," in *Proc. IEEE World Congr. Comput. Intell.*, 2022, pp. 18–23.
- [54] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evol. Comput.*, vol. 8, no. 2, pp. 173–195, Jun. 2000.
- [55] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, *Scalable Test Problems for Evolutionary Multiobjective Optimization*. London, U.K.: Springer, 2005, pp. 105–145.
- [56] R. Cheng et al., "A benchmark test suite for evolutionary many-objective optimization," *Complex Intell. Syst.*, vol. 3, pp. 67–81, Mar. 2017.
- [57] R. Cheng, Y. Jin, M. Olhofer, and B. Sendhoff, "Test problems for large-scale multiobjective and many-objective optimization," *IEEE Trans. Cybern.*, vol. 47, no. 12, pp. 4108–4121, Dec. 2017.

- [58] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem. “Brax—A differentiable physics engine for large scale rigid body simulation.” 2021. [Online]. Available: <http://github.com/google/brax>
- [59] G. Brockman et al. “OpenAI gym.” 2016. [Online]. Available: <https://openai.com/research/openai-gym-beta>
- [60] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proc. Int. Conf. Neural Netw.*, vol. 4, 1995, pp. 1942–1948.
- [61] R. Storn and K. Price, “Differential evolution—A simple and efficient heuristic for global optimization over continuous spaces,” *J. Global Optim.*, vol. 11, no. 4, pp. 341–359, Dec. 1997.
- [62] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, Rep. TR-2009, 2009. [Online]. Available: <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>
- [63] V. Nair and G. E. Hinton, “Rectified linear units improve restricted Boltzmann machines,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814.
- [64] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017, *arXiv:1707.06347*.
- [65] P. Dhariwal et al. “OpenAI baselines.” 2017. [Online]. Available: <https://github.com/openai/baselines>
- [66] M. Towers et al. “Gymnasium.” Mar. 2023. [Online]. Available: <https://zenodo.org/record/8127025>
- [67] A. Gupta, Y.-S. Ong, and L. Feng, “Multifactorial evolution: Toward evolutionary multitasking,” *IEEE Trans. Evol. Comput.*, vol. 20, no. 3, pp. 343–357, Jun. 2016.
- [68] K. C. Tan, L. Feng, and M. Jiang, “Evolutionary transfer optimization - a new frontier in evolutionary computation research,” *IEEE Comput. Intell. Mag.*, vol. 16, no. 1, pp. 22–33, Feb. 2021.



**Beichen Huang** (Graduate Student Member, IEEE) received the B.Eng. degree in computer science and technology from the Southern University of Science and Technology, Shenzhen, China, in 2023. He is currently pursuing the Ph.D. degree with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, SAR.

His main research interests include evolutionary algorithms, neuroevolution, and high-performance evolutionary computation.



**Ran Cheng** (Senior Member, IEEE) received the B.Sc. degree from Northeastern University, Shenyang, China, in 2010, and the Ph.D. degree from the University of Surrey, Guildford, U.K., in 2016.

He is currently an Associate Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China.

Dr. Cheng is a recipient of the 2018 and 2021 IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION Outstanding Paper Award, the 2019 IEEE Computational Intelligence Society (CIS) Outstanding Ph.D. Dissertation Award, and the 2020 *IEEE Computational Intelligence Magazine* Outstanding Paper Award. He is the Founding Chair of the IEEE CIS Shenzhen Chapter. He is an Associate Editor of IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, IEEE TRANSACTIONS ON ARTIFICIAL INTELLIGENCE, IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTATIONAL INTELLIGENCE, and IEEE TRANSACTIONS ON COGNITIVE AND DEVELOPMENTAL SYSTEMS.



**Zhuozhao Li** received the B.E. degree from Zhejiang University, Hangzhou, China, in 2010, the M.S. degree from the University of Southern California, Los Angeles, CA, USA, in 2012, and the Ph.D. degree from the University of Virginia, Charlottesville, VA, USA, in 2018.

He is an Assistant Professor with the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China. His research interests

span the broad areas of distributed systems, high-performance computing, and cloud computing, with the emphasis on designing the foundation methodologies to optimize system performance for efficient computing.

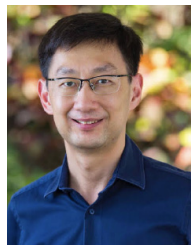


**Yaochu Jin** (Fellow, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees in automatic control from Zhejiang University, Hangzhou, China, in 1988, 1991, and 1996, respectively, and the Dr.-Ing. degree in neuroinformatics from Ruhr-University Bochum, Bochum, Germany, in 2001.

He is currently a Chair Professor of AI with the School of Engineering, Westlake University, Hangzhou, leading the Trustworthy and General AI Laboratory. Prior to that, he was an Alexander von Humboldt Professor of AI with the Faculty

of Technology, Bielefeld University, Bielefeld, Germany, and a Surrey Distinguished Chair of Computational Intelligence with the Department of Computer Science, University of Surrey, Guildford, U.K. He was a Finland Distinguished Professor in Finland and a Changjiang Distinguished Visiting Professor in China. His main research interests include data-driven evolutionary optimization, trustworthy machine learning, multiobjective evolutionary learning, and evolutionary developmental systems.

Dr. Jin is the recipient of the 2018, 2021, and 2023 IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION Outstanding Paper Award, and the 2015, 2017, and 2020 *IEEE Computational Intelligence Magazine* Outstanding Paper Award. He was named “Highly Cited Researcher” for 2019–2023 by Clarivate Analytics. He is currently the President of the IEEE Computational Intelligence Society and the Editor-in-Chief of *Complex & Intelligent Systems*. He is a member of Academia Europaea.



**Kay Chen Tan** (Fellow, IEEE) received the B.Eng. (First-Class Hons.) and Ph.D. degrees from the University of Glasgow, Glasgow, U.K., in 1994 and 1997, respectively.

He is currently a Chair Professor with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. He is also an Honorary Professor with the University of Nottingham, Nottingham, U.K., and the Chief Co-Editor of Springer Book Series on *Machine Learning: Foundations, Methodologies, and*

*Applications*.

Dr. Tan was the Editor-in-Chief of IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION from 2015 to 2020 and *IEEE Computational Intelligence Magazine* from 2010 to 2013, and currently serves as an editorial board member of 10+ journals. He is currently the Vice-President (Publications) of the IEEE Computational Intelligence Society.