Accelerating Bilevel Optimization with Hierarchical Many-threaded Parallel Differential Evolution

Amanda S. Dufek, Jaqueline S. Angelo, Douglas A. Augusto, and Helio J.C. Barbosa

***Abstract*—Bilevel optimization is encountered in many relevant real-world applications. The main feature of this type of problem is that an upper-level optimization problem is constrained by a nested lower-level optimization problem. Because of this nested structure, bilevel problems are usually computationally expensive to solve. Differential Evolution has demonstrated promising results in solving bilevel problems of relatively small scales. As the problem scale increases, the decision space becomes intrinsically larger, requiring a growing number of function evaluations for the method to work properly. In this context, heavy parallelization and high-performance computing techniques are indispensable to enable the resolution of more complex and challenging optimization problems. Hence, we propose a hierarchical many-threaded parallel Differential Evolution approach for bilevel problems, where both levels are parallelized. The computational experiments demonstrate that the parallel implementation achieved runtime speeds ranging from 44 to 2559 times faster than the sequential version on a well-known scalable SMD benchmark test problem when executed on an NVIDIA A100 GPU. The findings indicate that the algorithm's convergence is strongly influenced by the number of both upper- and lower-level generations. Moreover, the success of experiments with large-scale problems is closely linked to the choice of small population sizes.**

***Index Terms*—Bilevel Optimization, Differential Evolution, Hierarchical Decomposition, Many-threaded Parallelism, GP-GPU, Heterogeneous Computing**

I. Introduction

BILEVEL problems (BLPs) have gained significant attention in recent years. They address problems having a hierarchical structure involving two decision-makers: the *leader*, responsible for optimizing the upper-level problem, and the *follower*, responsible for the lower-level optimization problem. This nested structure is defined as a mathematical programming problem in which the leader has a subset of its variables constrained to be the optimal solution of the follower's optimization problem. Such an approach is useful to model decentralized planning problems, which arise in many real-world scenarios. The BLP can be defined as

$$\begin{aligned} &\min_{\mathbf{x}_{ul},\mathbf{x}_{ll}} F(\mathbf{x}_{ul},\mathbf{x}_{ll}) \\ &\text{subject to } G(\mathbf{x}_{ul},\mathbf{x}_{ll}) \leq 0 \\ &\qquad \mathbf{x}_{ll} \in \arg\min_{\mathbf{x}_{ll}} \{f(\mathbf{x}_{ul},\mathbf{x}_{ll}) : g(\mathbf{x}_{ul},\mathbf{x}_{ll}) \leq 0)\} \end{aligned}$$

where $\mathbf{x}_{ul} \in \mathbb{R}^n$ and $\mathbf{x}_{ll} \in \mathbb{R}^m$ are the control variables of the upper and the lower level, respectively, $F : \mathbb{R}^{n+m} \to \mathbb{R}$ is the upper-level objective function and $f : \mathbb{R}^{n+m} \to \mathbb{R}$ is the lower-level objective function. $G : \mathbb{R}^{n+m} \to \mathbb{R}^p$ and $g : \mathbb{R}^{n+m} \to \mathbb{R}^q$ denote the upper- and lower-level constraints, respectively. A BLP aims to identify the optimum value of the upper-level problem, subject to optimally solving the lower-level problem.

Compared with single-level optimization, BLPs introduce additional challenges due to their hierarchical structure. Even the "simplest" case of BLPs with linear functions on both levels may be non-linear [1] and were shown to be strongly NP-hard [2]. Just to identify if a solution is optimal or not is itself NP-hard [3]. Another challenge is that unless the lower-level solution is optimal, it cannot be considered feasible for the overall problem. This suggests that approximate methods, such as metaheuristics, may not be appropriate for solving the lower-level problem since they do not guarantee exact optimality. Nevertheless, near-optimal or satisfactory solutions — those that meet decision-maker's needs — are often acceptable in practice [4], particularly when the computational cost of obtaining an exact solution makes exact methods impractical.

Due to the complexity of BLPs, metaheuristics like Evolutionary Algorithms (EAs) have emerged as a promising alternative, as they do not require problem-specific assumptions such as linearity, convexity, or differentiability. EAs can also be applied in black-box optimization, where the objective function (or constraints) is derived from computer simulations or experimental data based on input values. Some reviews of applications and solution methods for BLPs, including both classical and evolutionary approaches, can be found in [5]–[7]. Notable works on EAs applied to BLPs can be roughly categorized into five groups:

1) Single-level reduction: Those techniques investigate the properties of the so-called inducible region to obtain (necessary and sufficient) conditions to replace the (convex) lower-level problem by its Karush-Kunh-Tucker (KKT) conditions, appending the resultant system to the upper-level problem. This approach results in a mathematical program with equilibrium constraints (MPEC), which is a single-level, but non-

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. This work was supported by the Brazilian agency Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) (grant number 301524/2023-8).

The author A.S. Dufek is with the National Energy Research Scientific Computing Center (NERSC), Berkeley, CA, USA (e-mail: asdufek@lbl.gov).

The authors J.S. Angelo and H.J.C. Barbosa are with the National Laboratory for Scientific Computing, Petrópolis, RJ, Brazil (e-mails: jsangelo@lncc.br, hcbm@lncc.br).

The author D.A. Augusto is with the Oswaldo Cruz Foundation, Rio de Janeiro, RJ, Brazil (e-mail: daa@fiocruz.br).

convex, optimization problem [8]. To apply such a strategy it is necessary to assume (explicit) knowledge of the analytical form of the objective function and constraints, and even so, sometimes the transformed problem is not equivalent to the original BLP [9], [10]. Although such an approach is common in classical methods [6], [11], there are fewer examples of this application in EAs [12]–[14]. Those methods are mostly limited to handling BLPs assuming linearity, convexity, differentiability, and a reduced number of variables.

2) Nested approaches: Unlike the previous strategy, nested approaches solve the bilevel problem in its original form by employing search processes at both levels in two different ways: (i) using EAs on the upper level and deterministic methods for the lower level [15], [16], and (ii) using EAs on both levels [16]–[20]. While these methods are the most straightforward and simple way to solve BLPs, the thorough lower-level searches for each upper-level variable, though capable of producing competitive results, are computationally expensive due to the high number of function evaluations.

3) Surrogate modeling: To reduce the computational effort associated with function evaluations, the use of surrogate modeling at one or both levels of bilevel optimization has gained popularity. Surrogate modeling aims to replace the original evaluation function with a substantially less expensive approximation. Nested surrogate-assisted methods, such as those proposed in [21]–[25], specifically focus on minimizing the computational effort at the lower level. Similarly, in [26]–[28], surrogate models are incorporated at both levels, while in [29], [30], surrogate modeling is applied to the upper-level optimization only. It is important to highlight that some of these methods are problem-specific approximate models that may not generalize well across different BLPs and may face convergence issues due to the accumulation of small deviations from the true optimum [31].

4) Information sharing: Another approach involves sharing information between tasks or problems to improve and accelerate the bilevel optimization procedure. This approach assumes that knowledge gained from solving one optimization problem can be used to accelerate the solution of a similar problem. Multitasking and transfer learning-based methods such as those in [30]–[36] employ information-sharing mechanisms between multiple lower-level tasks to improve the lower-level search and thus reduce the number of lower-level function evaluations. Note that, if tasks are not well-related, knowledge transfer can negatively impact optimization (negative transfer) [37], particularly in bilevel problems with different lower-level optimization tasks. They can also add complexity by requiring additional mechanisms to assess what knowledge should be transferred and how it should be applied [38].

5) Parallelism: The inherent parallelism of EAs is a propitious field for designing parallel techniques capable of exploring the maximum capacity of high-performance computing platforms, especially massively parallel ones, such as General-Purpose computing on Graphics Processing Units (GP-GPU) [39], to accelerate bilevel evolutionary optimization. The massively parallel architecture of GPUs makes them more efficient than general-purpose Central Processing Units (CPUs) when a large amount of independent data needs to be processed in parallel, which holds a promising opportunity for the resolution of BLPs. The scientific community in evolutionary computation has been engaged in constructing effective and efficient bilevel evolutionary techniques. It is interesting to note that while parallelism has been extensively investigated within EAs [40]–[48], its application in bilevel optimization remains relatively scarce. There are currently only a few examples of CPU-based parallel approaches applied to BLPs, where parallelism is implemented at either the upper-level [49], [50] or at both levels [51]–[54]. For instance, Bostian *et al.* [49] focused on bilevel multi-objective optimization for optimal policy in environmental economics. In [51], [52], the authors addressed scenarios involving one leader and multiple followers, which represents another challenging class of BLPs. Co-evolutionary decomposition-based algorithms, such as in [53], [54], were designed to solve combinatorial BLPs in supply chain management. Similarly, Camacho-Vallejo and Garcia-Reyes [50] introduced a co-evolutionary method for a binary BLP in telecommunication networks. To the best of our knowledge, GPU-based parallel approaches within the bilevel domain remains unexplored.

In this context, we propose a hierarchical, many-threaded, parallel differential evolution-based nested approach for BLPs, where both upper and lower-level search procedures are parallelized, designed to run on multiple parallel devices (e.g. CPUs and GPUs) from a variety of vendors (see Section III). Three parallel versions of the algorithm are described in Section IV-A3 and validated in Section IV-B. The corresponding sequential version developed by Angelo *et al.* [17] is presented in Section II. The motivation for parallelizing a nested approach lies in its simplicity and straightforwardness in solving BLPs. However, parallel hardware acceleration are not restricted to nested structures; it can also be applied to a range of previously mentioned algorithmic strategies [55].

Another contribution of this paper is providing a performance analysis of the proposed approach when applied to a well-known scalable SMD benchmark test problems [18] on an NVIDIA A100 GPU (see Section IV-C), taking into account up to 64 decision variables (see Section IV-D). We have not found in the existing literature any work that addresses the consideration of such a high number of variables in the context of the SMD test problems. An overview of the SMD benchmark test problem is given in Section IV-A2.

II. Differential Evolution for BLP

A. Differential Evolution Algorithm

Differential Evolution (DE) [56], [57] is a simple population-based stochastic method originally designed for solving optimization problems in continuous search spaces. DE and its variants have been successfully applied in numerous real-world problems from various domains. They are considered one of the most competitive and versatile members of the family of EAs [58].

A pseudo-code of the DE algorithm is shown in Algorithm 1. The method initiates with a random population $\mathbf{X}$ comprising N candidate solutions within the search space

Algorithm 1: DE algorithm

```
1  for i ← 1 to N do
2      x^i ← initiate_random_pop(x^(L), x^(U));
3      f_X[i] ← evaluate(x^i);
4  for g ← 1 to G do
5      for i ← 1 to N do
           // Eqs. (1-2)
6          v^i ← apply_operators(CR, F, x^{r1,r2,r3,i});
7          f_v ← evaluate(v^i);
8          if f_v ≤ f_X[i] then
9              x^i ← v^i; f_X[i] ← f_v;
10 select_best(f_X);
11 return the best solution found
```

defined by the specified lower bound, $x^{(L)}$, and upper bound, $x^{(U)}$, for the variable $\mathbf{x}$ (line 2). Each solution $\mathbf{x}^i$, where $i = 1, \ldots, N$, is evaluated based on an objective function (line 3), assigning a quality measure to the individuals.

At each generation g (line 4), each individual of population $\mathbf{X}$ undergoes mutation and crossover operations (line 6) following a predefined DE variant. For instance, consider the DE/target-to-rand/1/bin variant, described by Eqs. 1 and 2 below, as an example to demonstrate these two genetic operators.

In the mutation operation, for each vector $\mathbf{x}^i$ of size D, a vector $\mathbf{w}^i$ is generated according to:

$$w^i[j] = x^i[j] + F \times (x^{r_1}[j] - x^i[j]) + F \times (x^{r_2}[j] - x^{r_3}[j]) \quad (1)$$

where r_1-th, r_2-th, r_3-th are distinct and randomly selected individuals of $\mathbf{X}$, and different from the i-th; $j = 1, \ldots, D$, with D being the dimension of the problem; and F is the scale parameter used to control the amplitude of the search in the direction of the applied difference.

In addition, a crossover operation is performed, where the trial vector $\mathbf{v}^i$ is created using elements from both the target vector $\mathbf{x}^i$ and the donor vector $\mathbf{w}^i$ as:

$$v^i[j] = \begin{cases} w^i[j], & \text{if } \texttt{rand}(0,1) < CR \text{ or } j = jRand, \\ x^i[j], & \text{otherwise} \end{cases} \quad (2)$$

The crossover rate CR is a user-defined parameter, $\texttt{rand}(0, 1)$ is a uniformly distributed random number in the interval $[0, 1]$, and $jRand$ is a randomly generated integer from 1 to D so that $\mathbf{v}^i \neq \mathbf{x}^i$, $i = 1, \ldots, N$.

The fitness of the trial vector $\mathbf{v}^i$, calculated at line 7, is compared to that of the target vector $\mathbf{x}^i$ (line 8). The better solution is selected for the next generation of $\mathbf{X}$. Note that we are considering a minimization problem, thus, the lower the fitness, the better the solution. This evolutionary process continues until a stopping criterion is reached, in this case, the maximum number of generations G. Finally, the DE algorithm returns as a result the best solution found.

B. Bilevel Differential Evolution Algorithm

The most straightforward approach to applying DE in BLPs is to construct a nested procedure in which two DE algorithms are employed. Each DE algorithm is responsible for optimizing one level of the BLP: the upper-level DE (DE_{UL}) and the lower-level DE (DE_{LL}). A pseudo-code of the sequential version of the Bilevel Differential Evolution algorithm (BLDE) is given by Algorithm 2 [17]. The BLDE algorithm or the upper-level DE iteratively evolves pairs of solutions. For each upper-level solution $\mathbf{x}^i_{ul}$, where $i = 1, \ldots, N_{ul}$, there is an associated optimum lower-level solution $\mathbf{x}^{*^i}_{ll}$. To facilitate this process, BLDE maintains two populations: the upper-level population $\mathbf{X}_{ul}$, which consists of N_{ul} upper-level candidate solutions, and $\mathbf{X}^*_{ll}$, a collection of the corresponding N_{ul} optimum lower-level solutions obtained so far. The initialization and evaluation of both populations occur between lines 1 and 5.

Algorithm 2: BLDE algorithm or DE_{UL}

```
1  for i ← 1 to N_ul do
2      x^i_ul ← initiate_random_pop_Xul(x^(L)_ul, x^(U)_ul);
3      (x^{*^i}_ll, f_xll) ← DE_LL(x^i_ul);          // Algorithm 3
4      f_Xul[i] ← evaluate_ul(x^i_ul, x^{*^i}_ll);
5      f_X*ll[i] ← f_xll;
6  for g ← 1 to G_ul do
7      for i ← 1 to N_ul do
8          v^i_ul ← apply_operators_vul(CR, F, x^{r1,r2,r3,i}_ul);
9          (v^i_ll, f_vll) ← DE_LL(v^i_ul);           // Algorithm 3
10         f_vul ← evaluate_ul(v^i_ul, v^i_ll);
11         if f_vul ≤ f_Xul[i] and f_vll ≤ f_X*ll[i] then
12             x^i_ul ← v^i_ul; f_Xul[i] ← f_vul;
13             x^{*^i}_ll ← v^i_ll; f_X*ll[i] ← f_vll;
14 select_best(f_Xul, f_X*ll);
15 return the best pair of solutions found
```

At each generation g (line 6), each upper-level target vector $\mathbf{x}^i_{ul}$ is subjected to mutation and crossover operations to generate its respective upper-level trial vector $\mathbf{v}^i_{ul}$ (line 8). For each fixed upper-level trial vector $\mathbf{v}^i_{ul}$, $i = 1, \ldots, N_{ul}$, the lower-level DE is executed at line 9. The DE_{LL} procedure is outlined in Algorithm 3, which closely resembles Algorithm 1 from the previous section, except for the objective function (lines 3 and 7 of Algorithms 1 and 3). DE_{LL} returns the best lower-level solution $\mathbf{v}^i_{ll}$ from the lower-level population $\mathbf{X}_{ll}$ of size N_{ll}, obtained in response to the given $\mathbf{v}^i_{ul}$, based on the lower-level objective function. In the upper level, the pair of

Algorithm 3: DE_{LL}

```
   Input: v^i_ul
1  for j ← 1 to N_ll do
2      x^j_ll ← initiate_random_pop_Xll(x^(L)_ll, x^(U)_ll);
3      f_Xll[j] ← evaluate_ll(v^i_ul, x^j_ll);
4  for g ← 1 to G_ll do
5      for j ← 1 to N_ll do
6          v^j_ll ← apply_operators_vll(CR, F, x^{r1,r2,r3,j}_ll);
7          f_vll ← evaluate_ll(v^i_ul, v^j_ll);
8          if f_vll ≤ f_Xll[j] then
9              x^j_ll ← v^j_ll; f_Xll[j] ← f_vll;
10 select_best(f_Xll);
11 return the best solution found and its fitness
```

trial vectors, ($\mathbf{v}_{ul}^i$, $\mathbf{v}_{ll}^i$), is also evaluated based on the upper-level objective function (line 10). At line 11, its fitness in both upper and lower levels is compared to that of its respective pair of target vectors ($\mathbf{x}_{ul}^i$, $\mathbf{x}_{ll}^{*^i}$). The better pair of solutions is selected for the next generation of populations $\mathbf{X}_{ul}$ and $\mathbf{X}_{ll}^*$. At the upper and lower levels, their respective iterative procedure continues until their stopping criterion, given by a maximum number of upper- and lower-level generations, G_{ul} and G_{ll}, respectively, is satisfied. The BLDE algorithm concludes by returning the best-found pair of solutions. For more details the reader is referred to [17], [59]–[61].

III. Many-threaded Parallel BLDE

Taking as a baseline the previously introduced BLDE algorithm [17], we propose a hierarchical parallel implementation of BLDE, here referred to as PBLDE, that comes down to the nested execution of two kernels: `Parallel_Bilevel_Optimization`, and `Parallel_Solution_Evaluate`, as we can see in Algorithm 4. These two kernels will be described in

Algorithm 4: PBLDE algorithm

```
1 seed();
  // Initialization (first call)
2 Parallel_Bilevel_Optimization();
3 while stop criteria not met do
4 |   Parallel_Bilevel_Optimization();
5 |   Parallel_Solution_Evaluate();
6 select_best(f_{X_ul}, f_{X*_ll});
7 return the best pair of solutions found
```

detail in the next two sections. The PBLDE algorithm has been written in C/C++ and OpenCL, and is available at https://github.com/amandasd/blde. OpenCL stands for *Open Computing Language*, an open standard for programming parallel cross-platform heterogeneous computing systems, including CPU, GPUs, and other architectures such as DSPs and FPGAs [62]. A code written in OpenCL will run on any OpenCL-supported platform regardless of the architecture, although some tweaks might be necessary to achieve the maximum performance on a specific architecture or device. This means that the above mentioned parallel BLDE implementation runs on massively parallel GPUs but would also run straightaway on conventional multiple core CPUs, for instance.

The first line of Algorithm 4 calls the `seed` kernel, which is listed in Algorithm 5. It ensures that each work-item will

Algorithm 5: `seed` kernel

```
1 global_id ← get_global_id();
2 seed[global_id] ← lcg(s + global_id + 1);
```

have its own pseudo-random seed (also known as *state*) — and therefore its own sequence of pseudo-random numbers— by taking the return of a Linear Congruential Generator (LCG) over a given global seed s plus the work-item's index shifted by one (to avoid zeros). A LCG is one of most well-known algorithms for generating sequences of pseudo-random numbers. It is widely used due to its simplicity and efficiency [63]. Keeping separate seeds allows for parallel generation of pseudo-random numbers, where each work-item's seed is updated asynchronously at each call of LCG whenever the work-item requires a pseudo-random number.

A. Parallel Bilevel Optimization kernel

A general pseudo-code of the `Parallel_Bilevel_Optimization` kernel is outlined in Algorithm 6, while Fig. 1 presents an illustrative schema to aid in understanding its overall functioning.

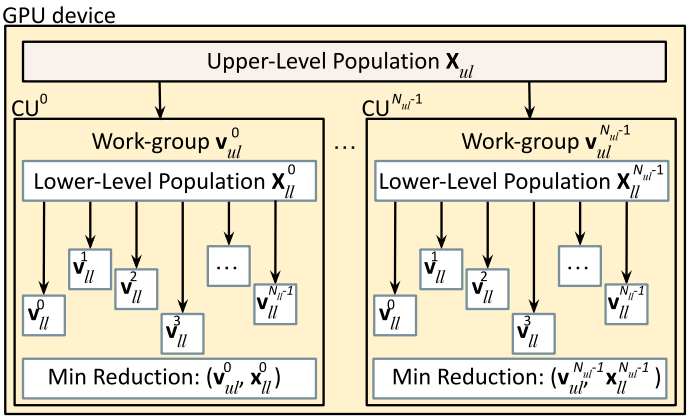

Fig. 1. An illustrative schema of the overall functioning of the `Parallel_Bilevel_Optimization` kernel on a GPU device.

A GPU device consists of hundreds of *compute units* (CUs), each composed of thousands of *processing elements* (PEs). The PBLDE algorithm is implemented by distributing the entire upper-level population across the compute units, while the lower-level population associated with each upper-level trial vector is spread among the processing elements within its respective compute unit (see Fig. 1). Each work-group — a group of work-items — runs within a single compute unit; however, each compute unit can execute simultaneously multiple work-groups. For this kernel, each work-group, indexed by $group_{id}$, corresponds to one upper-level trial vector $\mathbf{v}_{ul}^{group_{id}}$ of size D_{ul}, allocated in the work-group's local memory. The total number of work-groups is equivalent to the upper-level population size, N_{ul}, and the $local_{size}$ — number of work-items per work-group — is typically defined as the lower-level population size, N_{ll}. More details about the $local_{size}$ are provided at the end of this section.

The generation of $\mathbf{v}_{ul}^{group_{id}}$ by each work-group, executed by D_{ul} work-items, takes place between lines 3 and 10. The `barrier()` function at line 12 consists of a synchronization barrier for work-items within the same group. It guarantees that all work-items within the same group have consistent access to the vector $\mathbf{v}_{ul}^{group_{id}}$ in the remaining code. Upon the first call to the `Parallel_Bilevel_Optimization` kernel, the N_{ul} work-groups take care of the random initialization of the upper-level population, $\mathbf{X}_{ul}$, through the parallel execution of the `initiate_random_pop`$_{X_{ul}}$`()` function at line 5, where $x_{ul}^{(L)}$ and $x_{ul}^{(U)}$ are the lower and upper

Algorithm 6: `Parallel_Bilevel_Optimization` kernel

```
1  local_id ← get_local_id();
2  group_id ← get_group_id();
3  if first call then
       // Initialization of upper-level solution
4      if local_id < D_ul then
5          x_ul^{group_id}[local_id] ←
6                              initiate_random_pop_{x_ul}(x_ul^(L), x_ul^(U));
7          v_ul^{group_id}[local_id] ← x_ul^{group_id}[local_id];
8  else
       // Apply operators in upper-level solution
9      if local_id < D_ul then
10         v_ul^{group_id}[local_id] ←
11          apply_operators_{v_ul}(CR, F, x_ul^{r1,r2,r3,group_id}[local_id]);
12 barrier();
   // Lower-level optimization
13 x_ll^{local_id} ← initiate_random_pop_{x_ll}(x_ll^(L), x_ll^(U));
14 f_{X_ll}[local_id] ← evaluate_ll(v_ul^{group_id}, x_ll^{local_id});
15 for g ← 0 to G_ll do
16     for j ← 0 to D_ll do
17         v_ll^{local_id}[j] ←
18                apply_operators_{v_ll}(CR, F, x_ll^{r1,r2,r3,local_id}[j]);
19     barrier();
20     f_{v_ll} ← evaluate_ll(v_ul^{group_id}, v_ll^{local_id});
21     if f_{v_ll} ≤ f_{X_ll}[local_id] then
22         for j ← 0 to D_ll do
23             x_ll^{local_id}[j] ← v_ll^{local_id}[j]; f_{X_ll}[local_id] ← f_{v_ll};
24     barrier();
25 s ← 2^⌈log2(N_ll)⌉/2;
26 I[local_id] ← local_id;
27 while s > 0 do
28     barrier();
29     if local_id < s and local_id + s < N_ll then
30         if f_{X_ll}[I[local_id + s]] ≤ f_{X_ll}[I[local_id]] then
31             I[local_id] ← I[local_id + s];
32     s ← s/2;
33 if first call then
34     if local_id < D_ll then
35         X*_ll[group_id][local_id] ← x_ll^{I[0]}[local_id];
36 else
37     if local_id < D_ll then
38         V_ll[group_id][local_id] ← x_ll^{I[0]}[local_id];
39     if local_id < D_ul then
40         V_ul[group_id][local_id] ← v_ul^{group_id}[local_id];
```

bounds of variable $\mathbf{x}_{ul}$. The upper-level population contains N_{ul} upper-level target vectors $\mathbf{x}_{ul}^{group_{id}}$ of size D_{ul}, where $group_{id} = 0, \ldots, N_{ul} - 1$; in this case, $\mathbf{v}_{ul}^{group_{id}}$ is the same as $\mathbf{x}_{ul}^{group_{id}}$ (line 7). That is, there is a single upper-level population, allocated in the device's global memory, for all work-groups, whose individuals are identified by the superscript index in the variable $\mathbf{x}_{ul}$. For all the other calls to the `Parallel_Bilevel_Optimization` kernel, $\mathbf{v}_{ul}^{group_{id}}$ is created according to the `apply_operators`$_{v_{ul}}$`()` function, at line 10, in which the r_1-th, r_2-th and r_3-th are distinct and randomly selected individuals of $\mathbf{X}_{ul}$, and different from $group_{id}$-th.

The parallel DE_{LL} actually begins at line 13. Unlike the above, where just D_{ul} work-items are required, all the work-items — identified by the variable $local_{id}$ — per work-group are collectively responsible for two tasks:

(i) the random initialization of the lower-level population, $\mathbf{X}_{ll}$, through the parallel execution of the `initiate_random_pop`$_{x_{ll}}$`()` function at line 13, where $x_{ll}^{(L)}$ and $x_{ll}^{(U)}$ are the lower and upper bounds of variable $\mathbf{x}_{ll}$;

(ii) its subsequent evaluation based on the lower-level objective function, `evaluate`$_{ll}$`()` at line 14.

That is, each one of the N_{ul} work-groups is associated with a given upper-level trial vector, $\mathbf{v}_{ul}^{group_{id}}$, and its respective lower-level population, $\mathbf{X}_{ll}^{group_{id}}$. Each lower-level population contains N_{ll} lower-level target vectors $\mathbf{x}_{ll}^{local_{id}}$ of size D_{ll}, with $local_{id} = 0, \ldots, N_{ll} - 1$, allocated in the global memory of the device. Population $\mathbf{X}_{ll}$ is arranged in a transposed form — each solution row becomes a column in memory — to have coalesced memory accesses on GPUs. The same arrangement is applied to the other populations: $\mathbf{X}_{ul}$, $\mathbf{X}_{ll}^{*}$, $\mathbf{V}_{ul}$ and $\mathbf{V}_{ll}$; the last three will be explained below.

At each iteration g of line 15, each work-item — locally identified by $local_{id}$ — executes the following tasks:

(i) generates lower-level trial vectors $\mathbf{v}_{ll}^{local_{id}}$ of size D_{ll}, allocated in the device's local memory (lines 16–17);

(ii) compares the lower-level fitness relative to the pair ($\mathbf{v}_{ul}^{group_{id}}$, $\mathbf{v}_{ll}^{local_{id}}$) with that obtained by ($\mathbf{v}_{ul}^{group_{id}}$, $\mathbf{x}_{ll}^{local_{id}}$) (line 21); and

(iii) retains the better lower-level solution, $\mathbf{x}_{ll}^{local_{id}}$ or $\mathbf{v}_{ll}^{local_{id}}$, in $\mathbf{X}_{ll}^{group_{id}}$ (line 23).

That is, there is one $\mathbf{v}_{ll}^{local_{id}}$ for each work-item. This evolutionary process iterates until the stopping criterion, given by a maximum number of lower-level generations, G_{ll}, is satisfied. It is worth mentioning that the bilevel benchmark problem addressed in the current paper is an unconstrained minimization problem. For implementation purposes, there are two lower-level populations: one contains the target vectors, $\mathbf{x}_{ll}^{local_{id}}$, allocated in the device's global memory, and another contains the trial vectors, $\mathbf{v}_{ll}^{local_{id}}$, allocated in the device's local memory. At the end of each generation, both $\mathbf{X}_{ll}^{group_{id}}$ are identical and retain the best solutions.

The parallel reduction procedure for minimum operation takes place between lines 25 and 32 and identifies the best lower-level solution from $\mathbf{X}_{ll}^{group_{id}}$. A schematic representation of its execution is provided in Fig. 2 to facilitate understanding. At each iteration of the $while$ loop, $local_{size}$ work-items are simultaneously executed by each work-group. However, only the work-items identified by $local_{id} = 0, \ldots, \min(N_{ll} - s, s)$ execute line 30, which compares the lower-level fitness of the candidate solutions from $\mathbf{X}_{ll}^{group_{id}}$ indexed by $local_{id}$ and $local_{id} + s$, and stores the position of the smallest one in the $local_{id}$-th position of the array $\mathbf{I}$, allocated in the device's local memory. At the end of the parallel comparison, the first element of $\mathbf{I}$ contains the position associated with the best lower-level solution from

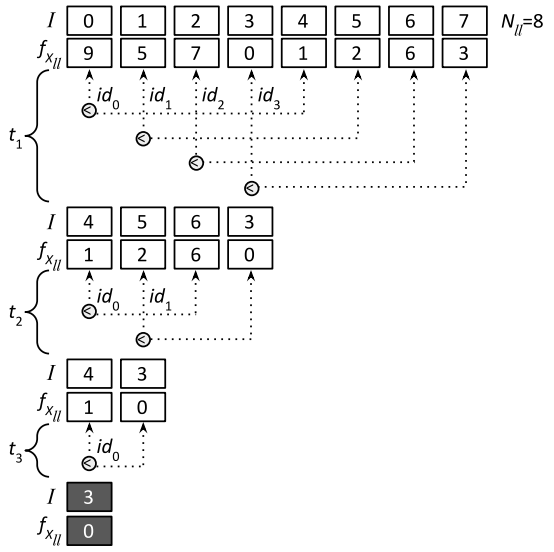

Fig. 2. A schematic representation of the execution of the parallel reduction procedure for minimum operation when $N_{ll} = 8$.

$\mathbf{X}_{ll}^{group_{id}}$. The function `barrier()` at line 28 guarantees the consistency of the local memory, since $I[local_{id} + s]$ refers to a memory region modified by a neighbor of the current $local_{id}$ work-item. For the optimal use of the parallel reduction, $local_{size}$ should be a power of 2. If $local_{size}$ is not a power of 2, the rounding to the next power of 2 in s calculation (line 25) implies $local_{id} + s > local_{size}$ for some ($local_{id}$, s) tuples. Thus, the checking at line 29 is required to avoid accessing invalid elements.

For each work-group, indexed by $group_{id}$, the best lower-level solution from $\mathbf{X}_{ll}^{group_{id}}$, identified by $I[0]$, is stored into array $\mathbf{V}_{ll}$ (line 38); and its upper-level trial vector, $\mathbf{v}_{ul}^{group_{id}}$, is stored into array $\mathbf{V}_{ul}$ (line 40). That is,$\mathbf{V}_{ul}$ contains N_{ul} upper-level trial vectors of size D_{ul}, and $\mathbf{V}_{ll}$ contains N_{ul} optimum lower-level solutions of size D_{ll}, where each pair ($\mathbf{v}_{ul}^{group_{id}}$, $\mathbf{x}_{ll}^{group_{id}}$) corresponds to one work-group. Upon the first call to the `Parallel_Bilevel_Optimization` kernel, the N_{ul} work-groups take care of the initialization of the population $\mathbf{X}_{ll}^{*}$ of size N_{ul} (line 35), whose main role is to store the optimum lower-level solutions obtained so far, as discussed in the next section. $\mathbf{X}_{ll}^{*}$, $\mathbf{V}_{ul}$ and $\mathbf{V}_{ll}$, allocated in the device's global memory, will be used by the `Parallel_Solution_Evaluate` kernel.

Given that the variables $\mathbf{X}_{ll}$, $\mathbf{f}_{X_{ll}}$, $\mathbf{I}$ and $\mathbf{v}_{ul}$ are allocated in the local memory of the device, the maximum N_{ll} and D_{ll} values will depend on the local memory capacity. For example, let an NVIDIA A100 GPU accelerator with a maximum local memory of 49,152 bytes, $D_{ul} = 2$, $D_{ll} = 3$ and double-precision floating-point type (8 bytes per scalar). The maximum N_{ll} value is 1,364, i.e.

$$\underbrace{1,364 \times 3 \times 8}_{\mathbf{X}_{ll}} \text{ bytes} + \underbrace{1,364 \times 8}_{\mathbf{f}_{X_{ll}}} \text{ bytes} + \underbrace{1,364 \times 4}_{\mathbf{I}} \text{ bytes} + \underbrace{2 \times 8}_{\mathbf{v}_{ul}} \text{ bytes} = 49,120 \text{ bytes}$$

An alternative approach to tackle the aforementioned challenge of dealing with large-dimension problems is based on the island model, in which the original lower-level population of N_{ll} members is divided into n GPUs, each one with a subpopulation size of $|N_{ll}|/n$. Each GPU or island would evolve its own population independently and periodically exchange individuals with other islands. That is, the `Parallel_Bilevel_Optimization` kernel would be executed simultaneously on multiple GPUs in a parallel cooperative way.

Due to device-specific hardware constraints, the $local_{size}$ — number of work-items per work-group — is defined as follows:

$$local_{size} = \begin{cases} local_{max\_size}, & \text{if } value \geq local_{max\_size} \\ & \text{and } N_{ll} \geq local_{max\_size}, \\ N_{ll}, & \text{if } value \geq local_{max\_size} \\ & \text{and } N_{ll} < local_{max\_size}, \\ N_{ll}, & \text{if } value < local_{max\_size} \\ & \text{and } value \geq N_{ll}, \\ value, & \text{if } value < local_{max\_size} \\ & \text{and } value < N_{ll} \end{cases} \quad (3)$$

whereas the $global_{size}$, which is the resulting total number of work-items (tasks):

$$global_{size} = N_{ul} \times local_{size}$$

ensuring that

(i) the maximum number of work-items per work-group, $local_{max\_size}$, provided by the computing device is not exceeded;

(ii) $local_{size}$ is not greater than N_{ll}, and

(iii) $global_{size}$ is divisible by $local_{size}$.

According to Eq. 3, $local_{size}$ can be less than N_{ll}. In this case, a given $local_{id}$ may cover multiple individuals of the lower-level population through an iterative procedure. Such iterative procedures have been omitted from the pseudo-code to facilitate the understanding of the parallel strategy. In addition, $local_{size}$ should be a multiple of the warp/wavefront size for improving hardware performance.

B. Parallel Solution Evaluate kernel

The `Parallel_Solution_Evaluate` kernel, described in Algorithm 7, was designed to accomplish two tasks:

(i) to evaluate the pairs of trial vectors, ($\mathbf{v}_{ul}^{global_{id}}$, $\mathbf{v}_{ll}^{global_{id}}$), stored in $\mathbf{V}_{ul}$ and $\mathbf{V}_{ll}$ by the `Parallel_Bilevel_Optimization` kernel, based on the both upper- and lower-level objective functions: `evaluate`$_{\text{ul}}$`()` and `evaluate`$_{\text{ll}}$`()` at lines 5 and 6, respectively.

(ii) to compare both fitness with those relative to its respective pair of target vectors, ($\mathbf{x}_{ul}^{global_{id}}$, $\mathbf{x}_{ll}^{*global_{id}}$). The better current pairs of solutions are integrated into the next upper-level generation of $\mathbf{X}_{ul}$ and $\mathbf{X}_{ll}^{*}$ (lines 7–9).

Each work-item — globally identified by $global_{id}$ — corresponds to a single pair of trial vectors, and $global_{size}$ denotes the total number of work-items, which in this case is N_{ul}. In the first call to the `Parallel_Solution_Evaluate`

Algorithm 7: `Parallel_Solution_Evaluate` kernel

```
global_id ← get_global_id();
2 if first call then
      f_{X_ul}[global_id] ← evaluate_ul(x_ul^{global_id}, x_ll^{*global_id});
4     f_{X*_ll}[global_id] ← evaluate_ll(x_ul^{global_id}, x_ll^{*global_id});
5 f_{v_ul} ← evaluate_ul(v_ul^{global_id}, v_ll^{global_id});
6 f_{v_ll} ← evaluate_ll(v_ul^{global_id}, v_ll^{global_id});
7 if f_{v_ul} ≤ f_{X_ul}[global_id] and f_{v_ll} ≤ f_{X*_ll}[global_id] then
      x_ul^{global_id} ← v_ul^{global_id}; f_{X_ul}[global_id] ← f_{v_ul};
9     x_ll^{*global_id} ← v_ll^{global_id}; f_{X*_ll}[global_id] ← f_{v_ll};
```

kernel, each work-item is responsible for evaluating one pair of target vectors, indexed by $global_{id}$, where $\mathbf{x}_{ul}^{global_{id}} \in \mathbf{X}_{ul}$ and $\mathbf{x}_{ll}^{*^{global_{id}}} \in \mathbf{X}_{ll}^{*}$ (lines 2–4). Note that the pairs ($\mathbf{x}_{ul}^{global_{id}}$, $\mathbf{x}_{ll}^{*^{global_{id}}}$) are the best ones obtained in the previous upper-level generation. The initialization (lines 5 and 35 of the Algorithm 6) and evaluation (lines 2–4 of the Algorithm 7) tasks related to $\mathbf{X}_{ul}$ and $\mathbf{X}_{ll}^{*}$ run only in the first call to the `Parallel_Bilevel_Optimization` and `Parallel_Solution_Evaluate` kernels, respectively. Thereafter, $\mathbf{X}_{ul}$ and $\mathbf{X}_{ll}^{*}$ are only updated according to lines 7–9 of the Algorithm 7. Unlike $\mathbf{X}_{ul}$ and $\mathbf{X}_{ll}^{*}$, a new $\mathbf{X}_{ll}^{group_{id}}$ is always generated by each one of the N_{ul} work-groups whenever `Parallel_Bilevel_Optimization` kernel executes (line 13 of Algorithm 6).

IV. Experiments

In this section, we begin by validating the parallel implementations of the BLDE algorithm against its existing sequential counterpart (see Section IV-B). Thereafter, a performance analysis of the PBLDE algorithm is presented in Section IV-C. Additionally, Section IV-D includes the application of the PBLDE algorithm to a more intricate and challenging optimization problem.

A. Experimental setup

1) Computational Environment: The parallel and sequential experiments were conducted, respectively, on a single NVIDIA A100 GPU and a 64-core AMD EPYC 7763 CPU ("Milan") running at 2.45 GHz (3.5 GHz boost). Each NVIDIA A100 GPU is equipped with 40 GB of global memory and a 40 MB L2 cache for the entire GPU, along with 108 compute units. Each compute unit has 192 KB of shared L1 cache and local memory, with a maximum of 2,048 processing elements and 65,536 registers. It accommodates work-group sizes of up to 1,024 work-items, organized into warps of 32 work-items each. Both BLDE versions were compiled with GNU GCC 11.2.0 with optimization flag `-O3`. The parallel version uses OpenCL and had its kernels built with the NVIDIA toolkit 11.7.

2) Benchmark and Parameters: A widely adopted benchmark for evaluating bilevel optimization methods is the scalable SMD test suite [18], which contains 12 test problems. The unconstrained SMD problems (SMD1-SMD8) are scalable in both the upper- and lower-level optimization dimensions. The objective functions at both levels contain configurable components, defined by parameters p, q, r, and s, which specify the respective dimensions. This introduces challenges in terms of convergence, complexity, and interaction (either conflict or cooperation) between the two levels. A comprehensive evaluation of the proposed PBLDE algorithm is carried out on the unconstrained SMD2 test problem. In SMD2, the lower-level objective function is convex and the upper-level task is convex with respect to upper-level variables and optimal lower-level variables. In this test problem, the upper- and lower-level problems are in conflict. This implies that, in the proximity of $\mathbf{x}_{ll}^{*}$ for a particular $\mathbf{x}_{ul}$, an improvement in the lower-level function value adversely affects the upper-level function value.

By capitalizing on the experiments performed to validate the PBLDE algorithm, we additionally conducted a sensitivity analysis of the key parameters that can potentially impact the performance of any optimizer when dealing with BLPs. Specifically, we investigated the influence of population size and the number of generations at both the upper and lower levels on performance, taking into account four dimensions of the SMD2 test problem using the following settings (for SMD2, $s = 0$):

- 2×3-variable version ($p = 1$, $q = 2$, and $r = 1$), where 2 is in the upper level and 3 is in the lower level.
- 5×5-variable version ($p = 3$, $q = 3$, and $r = 2$).
- 10×10-variable version ($p = 5$, $q = 5$, and $r = 5$).
- 32×32-variable version ($p = 16$, $q = 16$, and $r = 16$).

The BLDE's fixed parameters were set as follows: crossover rate $CR = 90\%$, scale factor $F = 80\%$, and a mutation operator based on DE/target-to-rand/1/bin. The choice of these parameters was based on experiments reported in previous works [17], [29], which tested various combinations during their preliminary exploratory phase and identified them as suitable options for the overall SMD test problems. Five population sizes expressed as powers of 2 ranging from 32 to 512 are considered at both upper and lower levels. Five lower-level number of generations are chosen between 50 and 500, except for the 32×32-variable version, where they are set at 700, 1000, 1200, and 1500 due to the increased problem difficulty. At the upper level, the stopping criteria are either a solution precision of 10^{-2} or a maximum of 10,000 generations.

Although we have randomly chosen a specific SMD benchmark test problem with a reasonable level of complexity for an in-depth analysis of the PBLDE algorithm while avoiding redundant experiments and waste of computational resources, its applicability can be extended to any unconstrained bilevel problem in general. For example, the PBLDE algorithm was also applied across the unconstrained SMD1 to SMD8 test problems for the two largest dimensions covered in this paper: 10×10 and 32×32, with population size of 32 in both levels and lower-level generation of 500 and 1500, respectively. More details can be found in the supplementary material.

3) PBLDE versions: The PBLDE algorithm, as described in Sections III-A and III-B, is essentially the parallel coun-

terpart of the original BLDE. The original BLDE is also called here the sequential version of BLDE, and the parallel version will be referred to as PBLDE *with barriers* henceforth. Two additional parallel versions have emerged through minor adjustments in the aforementioned implementation:

(i) A more flexible approach involves removing two non-critical synchronization barriers at lines 19 and 24 in Algorithm 6. This approach, referred here to as PBLDE *without barriers*, can potentially introduce a form of random mutation, as different work-items may require read access to lower-level target vectors — $\mathbf{x}_{ll}^{r_1}$, $\mathbf{x}_{ll}^{r_2}$, $\mathbf{x}_{ll}^{r_3}$, and $\mathbf{x}_{ll}^{local_{id}}$ — from $\mathbf{X}_{ll}$ allocated in the device's global memory at line 17 while they are being updated for the next generation at line 23.

(ii) The asynchronous version of PBLDE builds upon the previous one and increases flexibility by accessing the same r_1, r_2, r_3, and $local_{id}$ positions as before at line 17, but from $\mathbf{X}_{ll}$ allocated in the device's local memory rather than the device's global memory. It implies that the newly generated and not yet evaluated trial vectors, $\mathbf{v}_{ll}$, may be selected for participating in the crossover and mutation operators for creating the next lower-level population. This version is designed to reduce the higher latency reads from global memory, thereby improving the overall throughput.

B. Validation Analysis

Fig. 3 shows the number of upper-level generations required by the four versions of the BLDE algorithm to obtain the upper-level optimal solution to the SMD2 test problem with a solution precision of 10^{-2} at the upper level as a function of upper- and lower-level dimensions: 2×3, 5×5, 10×10. In case a given run reaches the maximum number of upper-level generations set to 10,000 and does not obtain the optimal solution, it is marked as failed. The number of failed runs is displayed at the top of the graph. Thirty independent runs were performed for each combination between upper- and lower-level population sizes, and number of lower-level generations. It accounts for a sample of 30 runs$\times$5 upper-level population sizes$\times$5 lower-level population sizes$\times$5 number of lower-level generations = 3,750 values plotted together as box-plots for each x-value and version of the BLDE algorithm. It is worth mentioning that only the non-failed runs were taken into account for the plots. An immediate outcome that emerges from Fig. 3 is a growing need for a higher number of upper-level generations as the scale of the problem increases. According to Fig. 3, the PBLDE algorithm with and without barriers show great similarity to the sequential BLDE algorithm in terms of convergence towards the optimal solution, with equivalent interquartile ranges, medians, whisker lengths, and number of failed runs. The SMD1 to SMD8 experiments in the supplementary material further support these findings (see top graph in Fig. S1). This corroborates the hypothesis that the two former PBLDE algorithms are on par with the sequential version's algorithmic effectiveness, which qualify us to proceed with a computational performance analysis of the PBLDE algorithm. Furthermore, it means that the possible random mutation introduced by removing the two non-critical synchronization barriers does not adversely affect the convergence rate to the optimal solution of the PBLDE algorithm. With respect to the computational performance, the difference between the PBLDE algorithm with and without barriers are not negligible, with median runtimes 14.3% (2×3), 9.1% (5×5), and 5.4% (10×10) faster for the latter (not shown). On the other hand, the asynchronous PBLDE algorithm exhibits a notably higher number of failed runs as the problem size increases. This indicates that the convergence towards the optimal solution may slow down or even stagnate when lower-quality candidate solutions are available for selection during reproduction. Moreover, this effect can worsen with more complex and challenging problems. Building on the information mentioned above, the discussion and analysis in the following sections will be centered on the PBLDE algorithm without barriers.

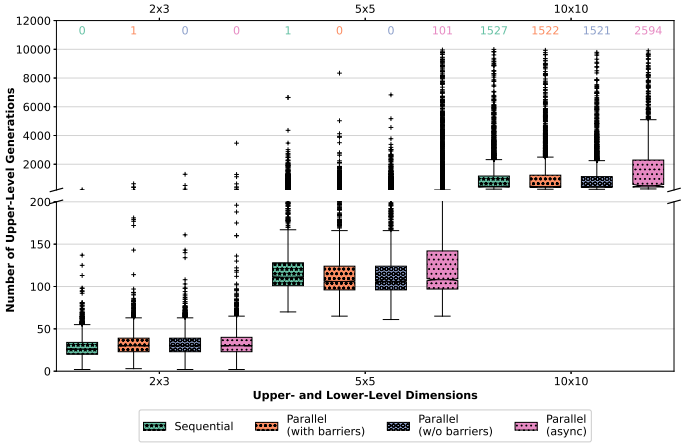

Fig. 3. Number of upper-level generations required by the four versions of the BLDE algorithm to obtain the upper-level optimal solution to the SMD2 test problem with a solution precision of 10^{-2} at the upper level as a function of upper- and lower-level dimensions: 2×3, 5×5, 10×10. The number of failed runs over a sample of size 3,750 is displayed at the top of the graph.

Fig. 4 shows an in-depth analysis of the failed runs for the four versions of the BLDE algorithm on the 10×10-variable SMD2 test problem as a function of three parameters: upper- and lower-level population sizes, and number of lower-level generations. It reveals that the 10×10-variable SMD2 test problem is notably influenced by the number of lower-level generations. The BLDE algorithm consistently discovers the bilevel optimal solution in all 30 independent runs for all population sizes, but only when the lower-level generations are 300 or higher. Although a similar trend remains valid for lower-level generations of 200 across most cases of population sizes, it requires, on average, 2–10 times more upper-level generations to find the best-known solution compared to lower-level generations set at 300 or higher (not shown). By consequence, it takes up to about 6 times more time to complete the execution (not shown). This slowdown in convergence in terms of upper-level generations, as well as the occurrence of failed runs, results from non-optimal parameter settings, such as an insufficient number of lower-level generations. Once the lower-level generation is appropriately

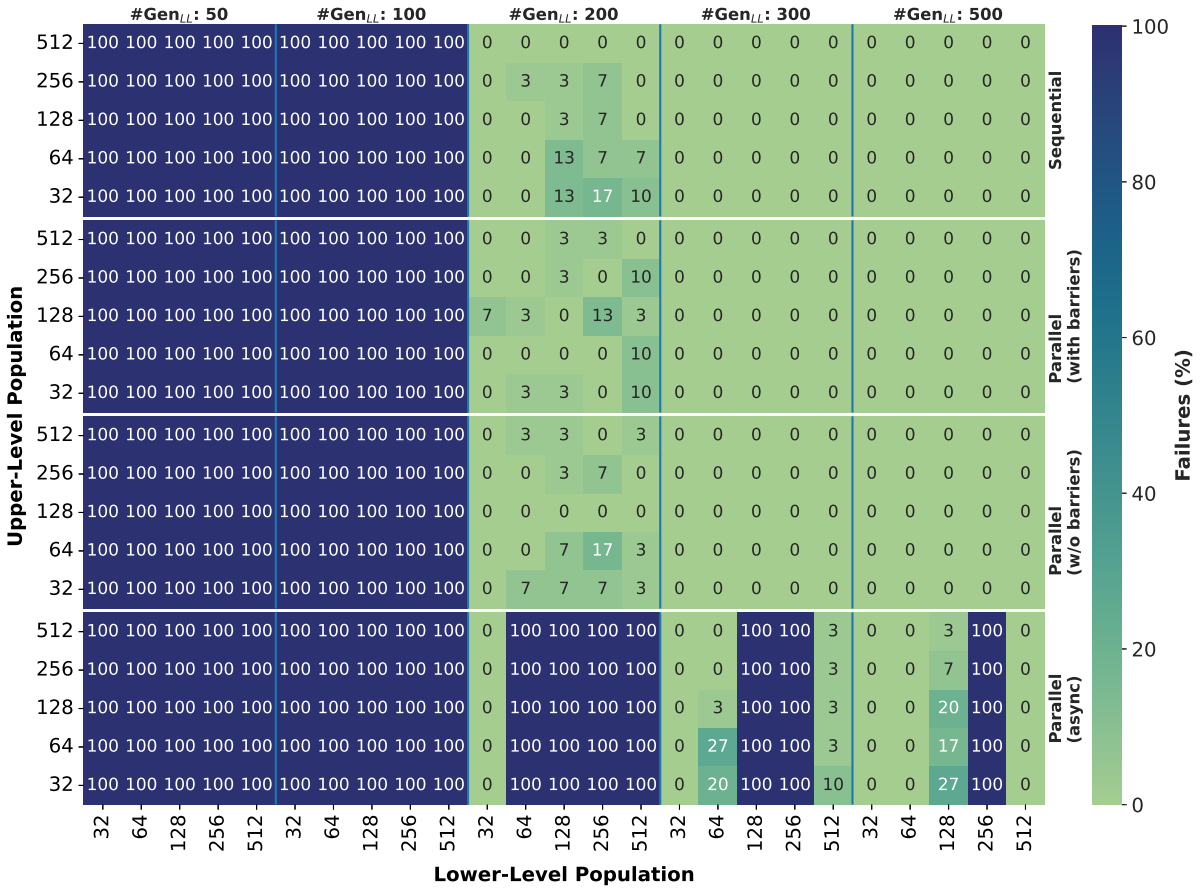

Fig. 4. The percentage of failed runs over a sample of size 30 for the four BLDE versions on the 10×10-variable SMD2 test problem as a function of three parameters: upper- and lower-level population sizes, and number of lower-level generations.

set — ensuring no failed runs — the number of upper-level generations required for an optimal solution shows low sensitivity to changes in population size, with average values ranging from 350 to 486 (not shown). Unlike the three upper versions in Fig. 4, the asynchronous version of PBLDE, which employs the most flexible strategy and exhibits the worst performance, requires more than 500 lower-level generations to achieve success across all population sizes. The above-mentioned results underscore the importance of effectively addressing the lower-level optimization problem to achieve a good quality solution at the upper level. They also highlight the need to properly adjust the number of lower-level generations according to the complexity of the target problem and the BLDE algorithm. Additionally, they back up the validation of the PBLDE algorithms with and without barriers. The performance drop for lower-level population sizes of 128 and 256, observed, for example, in the third column of Fig. 4 for the sequential version and even more pronounced in the last two columns of the asynchronous version of PBLDE, indicates an anomalous behavior in the evolutionary algorithm dynamics — persistent across different BLDE implementations — that impair convergence efficiency. Further experiments would be necessary to better understand these non-linear parameter interactions and their impact on performance.

C. Speedup analysis of low-dimension test problems

As we can see in Algorithm 4, these two startup tasks precede the iterative execution of the main kernels: determining the seed of each work-item from a pseudo-random number generator (line 1), to ensure that each work-item will generate its own pseudo-random number sequence; and the first call to the `Parallel_Bilevel_Optimization` kernel (line 2), which is responsible for setting up $\mathbf{X}_{ul}$ and $\mathbf{X}^*_{ll}$. The `OpenCL` runtime initialization also belongs to the startup tasks, which include the platform and device definition, kernels creation, as well as memory and data management. However, the startup tasks run only once at the beginning of the BLDE algorithm, and therefore will not be discussed in detail within this paper. The primary focus here is on the core of the BLDE algorithm, specifically the iterative execution of the two kernels (lines 4 and 5).

Fig. 5 shows the speedup of the `Parallel_Bilevel_Optimization` and `Parallel_Solution_Evaluate` kernels together corresponding to the PBLDE algorithm without barriers over its sequential version applied to the SMD2 test problem with 2×3, 5×5, and 10×10 dimensions as a function of three parameters: upper- and lower-level population sizes, and number of lower-level generations on an NVIDIA A100 GPU. The kernel runtime is determined from a sample

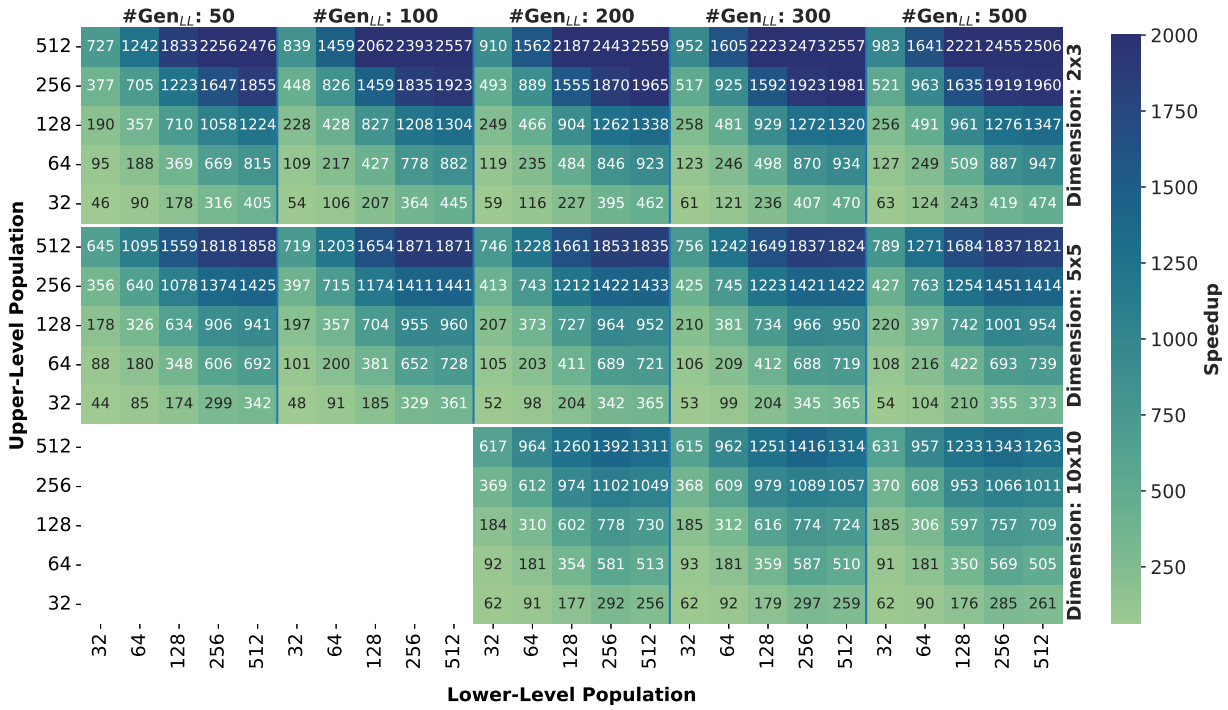

Fig. 5. Speedup of the `Parallel_Bilevel_Optimization` and `Parallel_Solution_Evaluate` kernels together corresponding to the PBLDE algorithm without barriers over its sequential version applied to the SMD2 test problem with 2×3, 5×5, and 10×10 dimensions as a function of three parameters: upper- and lower-level population sizes, and number of lower-level generations on an NVIDIA A100 GPU. Only non-failed runs were considered when calculating the speedup. In the absence of data, a blank cell is displayed.

mean over multiple kernel executions, followed by a sample median from 30 independent runs of the algorithm. It is worth mentioning that only non-failed runs were taken into account for calculating the speedup. As shown in Fig. 4 from the previous section, there were no successful sequential and parallel runs of the 10×10-variable SMD2 test problem when configured with lower-level generations of size 50 and 100. Consequently, blank cells are displayed in the current graph. From Fig. 5, it can be observed that the speedup of the `Parallel_Bilevel_Optimization` and `Parallel_Solution_Evaluate` kernels together over the corresponding sequential version increases with the population sizes, but decreases with the dimension of the problem, while it does not vary much with lower-level generation. The results demonstrate a substantial performance improvement of the PBLDE algorithm over its sequential version, with speedup values ranging from 46 to 2559, from 44 to 1871, and from 46 to 1416 for dimensions of 2×3, 5×5, and 10×10, respectively. For instance, the average total runtime decreases from 3 hours and 40 minutes to just 8 seconds with the following configuration: upper-level population size of 512, lower-level population size of 128, lower-level generation of 200, and 10×10 dimension. This represents an impressive reduction of 99.9% in the total execution time of the BLDE algorithm. Besides, this suggests that the PBLDE algorithm is capable of tackling more complex problems that were previously considered infeasible due to their high computational cost, as illustrated by the analysis of the 32×32-variable SMD2 test problem in the next section.

Algorithmic strategies for approximating optimal solutions in bilevel problems are widely used as an alternative to the development of parallel bilevel algorithms. In this context, the bilevel literature has proposed various surrogate-assisted approaches, along with novel methods based on transfer learning and multitasking paradigm, aimed at reducing the number of function evaluations and potentially saving computational time, despite the overhead introduced by incorporating these mechanisms into the search process. Recent and notable algorithmic strategies, such as BLEAQ [22], BLEAQ-II [25], ε-KKT [24], SABO [27], BLDES [29], M-BLEA [32], MT-BLDE [30], TLEAs [31], and BL-CMA-ES [33], have typically achieved optimal convergence on the SMD1-SMD8 test problems at speeds 2 to 7 times faster, in terms of the number of function evaluations, than their baseline benchmarks. However, reduced computational effort does not always translate into runtime speedup due to the potential overhead involved. On the other hand, as shown in the supplementary material, the PBLDE algorithm achieved runtime speeds at least 61 times faster than its sequential version on the SMD1-SMD8 test problems when executed in parallel on a single GPU, with both parallel and sequential versions employing equivalent computational effort (see top graph in Fig. S1). Fortunately, these two orthogonal optimization strategies — algorithmic efficiency and parallel hardware acceleration — can be effectively combined to leverage the advantages of both, further enhancing overall performance.

The existing CPU-based parallel algorithms were primarily developed for specific problems, such as multi-objective BLPs

for optimal policy in environmental economics [49], combinatorial BLPs in supply chain management [53], [54], and binary BLP in telecommunication networks [50]; therefore, they cannot be directly compared with the PBLDE algorithm on the SMD benchmark test problems.

D. Performance analysis of the 32×32*-variable test problem*

In this section we present an analysis of the parallel implementation of the BLDE algorithm when applied to the 32×32-variable SMD2 test problem. To the best of our knowledge, there is no other research reporting any results involving 32×32-variable SMD benchmark test problems. This may be attributed to the high computational cost associated with such problems, consequently making it infeasible to run them sequentially. Even for the PBLDE implementation, this 32×32-variable problem exceeded the local memory capacity of the high-end NVIDIA A100 GPU when we attempted to run with lower-level populations above 128 (reason explained in Section III-A), which fortunately was not an obstacle to achieving optimal convergence.

The percentage of failed runs plotted in Fig. 6 reinforces the earlier findings from Fig. 4 and provides new insights into convergence and the interaction between the two levels. The PBLDE algorithm's performance in discovering the bilevel optimal solution is closely linked not only to the precise definition of the number of lower-level generations, but also to the lower- and upper-level population sizes. Furthermore, it underscores the necessity of increasing the number of lower-level generations as the complexity of the optimization problems rises, going from 50 generations for 2×3 dimension to 300 for 10×10 and 1200 for 32×32. As depicted in Fig. 6, a distinct preference for small population sizes is evident. Successful runs were observed only when configured with a lower-level population size of 32 and an upper-level population size of 128 or lower. In particular, employing

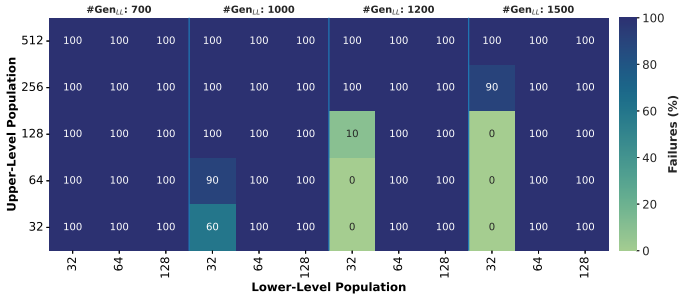

Fig. 6. The percentage of failed runs over a sample of size 10 for the PBLDE algorithm without barriers applied to the 32×32-variable SMD2 test problem as a function of three parameters: upper- and lower-level population sizes, and number of lower-level generations.

a population size of 32 in both levels leads to the fastest convergence to the optimal solution in terms of runtime and number of upper-level generations. The average total runtime relative to upper-level population sizes of 32, 64, and 128 takes approximately 50 seconds, 100 seconds, and 200 seconds, respectively. These execution times show little variation between lower-level generations of 1200 and 1500. With an increase in both the scale of the problem and the population size, the convergence speed towards the bilevel solution decreases. By consequence, more upper-level generations are required to obtain the optimal solution (see Fig. 3). This implies that the maximum number of upper-level generations must be set to over 10,000 when dealing with large population sizes, such as exceeding 128 individuals in the upper level, for successful convergence in the context of the 32×32-variable SMD2 test problem. An additional set of experiments was conducted using a smaller value for the scale factor ($F = 40\%$) with a lower-level generation of 1500; however, none of them resulted in successful runs.

Additionally, the `Parallel_Bilevel_Optimization` runtime tends to asymptotically dominate over that of `Parallel_Solution_Evaluate` with increasing problem dimension, as shown in Fig. 7. Despite the growing

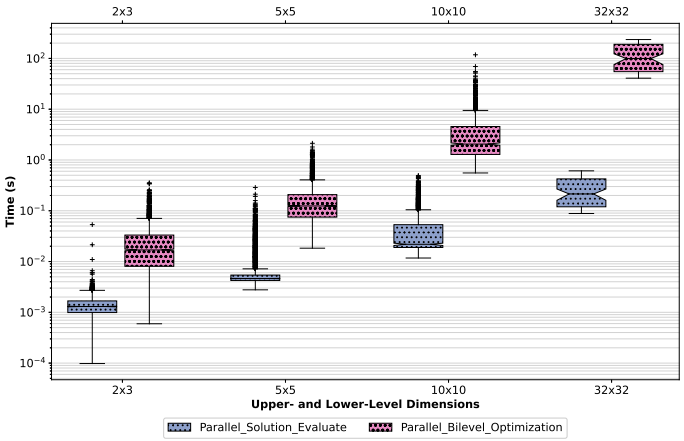

Fig. 7. Total time spent on each of the two kernels: `Parallel_Solution_Evaluate` and `Parallel_Bilevel_Optimization` for four dimensions of the SMD2 test problem: 2×3, 5×5, 10×10, and 32×32. It is worth mentioning that all successful runs were plotted together as box-plots for each x-value and kernel. Note that y-axis represents the execution time using a base-10 logarithmic scale.

execution time for both kernels as the scale of the problem increases, the former kernel exhibits a steeper positive slope and gradually diverges further from the latter kernel. For instance, the total time spent on the former kernel surpasses that on the latter kernel by more than 400 times for the 32×32 dimension. This discrepancy arises from the combined effect of higher numbers of upper-level generations and longer individual execution times for the `Parallel_Bilevel_Optimization` kernel. In contrast, the behavior of the `Parallel_Solution_Evaluate` kernel is primarily driven by higher numbers of upper-level generations, as its time for a single execution remains nearly constant (∼50 microseconds) regardless of parameters and problem complexity.

V. Conclusions

In this article we proposed a massively parallel implementation of a differential evolution-based bilevel optimization algorithm (BLDE), which we referred to as PBLDE. As far as we know,

this is the first parallel implementation of a BLDE for many-threaded architectures such as GPUs, on top of being portable across compute architectures.

We devised and investigated a standard version of PBLDE (*with barriers*) plus two increasingly relaxed/faster variations: *without barriers* and *async*. While the latter showed to degrade the success rate of the optimization due to the allowance of not-so-good candidate solutions during the recombination phase, the former proved to be as good as the standard (P)BLDE implementation but considerably faster than PBLDE *with barriers* (5.4%–14.3%).

Thanks to the aggressive two-level decomposition strategy, where both upper and lower levels were parallelized, and to the design thought to optimally explore modern computing architectures, PBLDE was able to run 44–2559 times faster on an NVIDIA A100 GPU than the sequential counterpart running on CPU, depending on the parameters chosen, for all three reference problem dimensions (2×3, 5×5, and 10×10 variables). Moreover, for the first time ever reported in the literature, PBLDE enabled a 32×32-variable bilevel problem to be solved feasibly, requiring just 50 seconds to converge.

Besides the remarkable performance of PBLDE, some additional conclusions and insights can be drawn from this research. Firstly, it became evident that a higher number of upper-level generations are desirable as the scale of the problem increases. This also holds true for the number of lower-level generations, which went from just 50 generations (for the 2×3-variable problem) up to 1200 generations (32×32-variable). When the number of lower-level generations is set too low, two things happen: (i) a higher number of upper-level generations is required; (ii) the overall runtime increases. Finally, our experiments clearly pointed out that small population sizes are in general preferred, resulting in both higher success rates and faster convergence. In particular, for the 32×32-variable problem, a population size of 32 individuals for both levels led to the fastest convergence to optimal solution in terms of runtime and number of upper-level generations.

For future research directions, we highlight the following endeavors: (i) adapt PBLDE to tackle the *constrained* version of bilevel optimization, and also (ii) the *multiple independent followers* variation. Both adaptions should require minor modifications to PBLDE. Likewise, another interesting work is to extend PBLDE to deal with *multiple-objective* bilevel optimization.

REFERENCES

[1] J. F. Bard, *Practical Bilevel Optimization*. Kluwer Academic Publisher, 1998.

[2] P. Hansen, B. Jaumard, and G. Savard, "New branch-and-bound rules for linear bilevel programming," *SIAM J. on Scientific and Statistical Computing*, vol. 13, no. 5, pp. 1194–1217, 1992.

[3] L. N. Vicente, G. Savard, and J. J. Júdice, "Descent approaches for quadratic bilevel programming," *J. of Optimization Theory and Applications*, vol. 81, no. 2, pp. 379–399, May 1994.

[4] K. Deb and A. Sinha, "Solving bilevel multi-objective optimization problems using evolutionary algorithms," in *Evolutionary Multi-Criterion Optimization*, M. Ehrgott, C. M. Fonseca, X. Gandibleux, J.-K. Hao, and M. Sevaux, Eds. Springer Berlin Heidelberg, 2009, pp. 110–124.

[5] A. Sinha, P. Malo, and K. Deb, "A review on bilevel optimization: From classical to evolutionary approaches and applications," *IEEE Trans. on Evolutionary Computation*, vol. 22, no. 2, pp. 276–295, 2018.

[6] S. Dempe and A. Zemkoho, *Bilevel Optimization: Advances and Next Challenges*. Cham: Springer Int. Publishing, 2020.

[7] J.-F. Camacho-Vallejo, C. Corpus, and J. G. Villegas, "Metaheuristics for bilevel optimization: A comprehensive review," *Computers & Operations Research*, vol. 161, p. 106410, 2024.

[8] Z.-Q. Luo, J.-S. Pang, and D. Ralph, *Mathematical Programs with Equilibrium Constraints*. Cambridge University Press, 1996.

[9] S. Dempe, *Foundations of Bilivel Programming*. Kluwer Academic Publisher, 2002.

[10] A. Chinchuluun, P. Pardalos, and H.-X. Huang, "Multilevel (hierarchical) optimization: Complexity issues, optimality conditions, algorithms," in *Advances in Applied Mathematics and Global Optimization*, ser. Advances in Mechanics and Mathematics, D. Y. Gao and H. D. Sherali, Eds. Springer US, 2009, vol. 17, pp. 197–221.

[11] M. S. Sapre and I. R. Kale, *A Brief Review of Bilevel Optimization Techniques and Their Applications*. Singapore: Springer Nature Singapore, 2023, pp. 1–24.

[12] G. Wang, Z. Wan, X. Wang, and Y. Lv, "Genetic algorithm based on simplex method for solving linear-quadratic bilevel programming problem," *Computers & Mathematics with Applications*, vol. 56, no. 10, pp. 2550–2555, 2008.

[13] Y. Jiang, X. Li, C. Huang, and X. Wu, "Application of particle swarm optimization based on chks smoothing function for solving nonlinear bilevel programming problem," *Applied Mathematics and Computation*, vol. 219, no. 9, pp. 4332–4339, 2013.

[14] Z. Wan, L. Mao, and G. Wang, "Estimation of distribution algorithm for a class of nonlinear bilevel programming problems," *Information Sciences*, vol. 256, pp. 184–196, 2014, business Intelligence in Risk Management.

[15] H. I. Calvete, C. Galé, and J. A. Iranzo, "Planning of a decentralized distribution network using bilevel optimization," *Omega*, vol. 49, pp. 30–41, 2014.

[16] J. S. Angelo and H. J. C. Barbosa, "A study on the use of heuristics to solve a bilevel programming problem," *Int. Transactions in Operational Research*, vol. 22, no. 5, pp. 861–882, 2015.

[17] J. S. Angelo, E. Krempser, and H. J. C. Barbosa, "Differential evolution for bilevel programming," in *IEEE Congress on Evolutionary Computation*, 2013, pp. 470–477.

[18] A. Sinha, P. Malo, and K. Deb, "Test problem construction for single-objective bilevel optimization," *Evolutionary Computation*, vol. 22, no. 3, pp. 439–477, 2014.

[19] L. Zhao and J. Wei, "A nested particle swarm algorithm based on sphere mutation to solve bi-level optimization," *Soft Computing*, vol. 23, pp. 11 331–11 341, 2019.

[20] J.-F. Camacho-Vallejo and D. Dávila, "Nested evolutionary algorithms for solving a bi-level warehouse location problem that considers inventory decisions," *Annals of Operations Research*, vol. 1, pp. 1–34, 2024.

[21] J. S. Angelo, E. Krempser, and H. J. Barbosa, "Differential evolution assisted by a surrogate model for bilevel programming problems," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 1784–1791.

[22] A. Sinha, P. Malo, and K. Deb, "An improved bilevel evolutionary algorithm based on quadratic approximations," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 1870–1877.

[23] M. M. Islam, H. K. Singh, and T. Ray, "A surrogate assisted approach for single-objective bilevel optimization," *IEEE Trans. on Evolutionary Computation*, vol. 21, no. 5, pp. 681–696, 2017.

[24] A. Sinha, T. Soun, and K. Deb, "Using karush-kuhn-tucker proximity measure for solving bilevel optimization problems," *Swarm and Evolutionary Computation*, vol. 44, pp. 496–510, 2019.

[25] A. Sinha, Z. Lu, K. Deb, and P. Malo, "Bilevel optimization based on iterative approximation of multiple mappings," *Journal of Heuristics*, vol. 26, no. 2, pp. 151–185, 2020.

[26] M. M. Islam, H. K. Singh, and T. Ray, "Efficient global optimization for solving computationally expensive bilevel optimization problems," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, 2018, pp. 1–8.

[27] J.-A. Mejía-de Dios and E. Mezura-Montes, "A surrogate-assisted metaheuristic for bilevel optimization," in *Proc. of the 2020 Genetic and Evolutionary Computation Conference*, ser. GECCO'20. Association for Computing Machinery, 2020, p. 629–635.

[28] H. Jiang, K. Chou, Y. Tian, X. Zhang, and Y. Jin, "Efficient surrogate modeling method for evolutionary algorithm to solve bilevel optimization problems," *IEEE Transactions on Cybernetics*, vol. 54, no. 7, pp. 4335–4347, 2024.

[29] J. S. Angelo, E. Krempser, and H. J. C. Barbosa, "Performance evaluation of local surrogate models in bilevel optimization," in *Machine Learning, Optimization, and Data Science*, G. Nicosia, P. Pardalos, R. Umeton, G. Giuffrida, and V. Sciacca, Eds. Cham: Springer Int. Publishing, 2019, pp. 347–359.

[30] I. L. Russo and H. J. Barbosa, "A multitasking surrogate-assisted differential evolution method for solving bi-level optimization problems," in *2022 IEEE Congress on Evolutionary Computation (CEC)*, 2022, pp. 1–8.

[31] L. Chen, H.-L. Liu, K. C. Tan, and K. Li, "Transfer learning-based parallel evolutionary algorithm framework for bilevel optimization," *IEEE Transactions on Evolutionary Computation*, vol. 26, no. 1, pp. 115–129, 2022.

[32] A. Gupta, J. Mańdziuk, and Y.-S. Ong, "Evolutionary multitasking in bi-level optimization," *Complex & Intelligent Systems*, vol. 1, no. 1, pp. 83–95, 2015.

[33] P.-Q. Huang, Q. Zhang, and Y. Wang, "Bilevel optimization via collaborations among lower-level optimization tasks," *IEEE Transactions on Evolutionary Computation*, vol. 27, no. 6, pp. 1837–1850, 2023.

[34] B. Wang, H. K. Singh, and T. Ray, "Investigating neighborhood solution transfer schemes for bilevel optimization," in *2022 IEEE Congress on Evolutionary Computation (CEC)*, 2022, pp. 1–8.

[35] L. Lin, T. Liu, H. Zhang, J. Leng, L. Wei, and Q. Liu, "Combined knowledge transfer and adaptive coordinate systems approach for evolutionary bilevel optimization," *Expert Systems with Applications*, vol. 228, p. 120309, 2023.

[36] B. Wang, H. K. Singh, and T. Ray, "An evaluation of simple solution transfer strategies for bilevel multiobjective optimization," in *2023 IEEE Congress on Evolutionary Computation (CEC)*, 2023, pp. 1–8.

[37] A. Gupta and Y.-S. Ong, "Back to the roots: Multi-x evolutionary computation," *Cognitive Computation*, vol. 11, pp. 1–17, 2019.

[38] A. Gupta, Y.-S. Ong, and L. Feng, "Insights on transfer optimization: Because experience is the best teacher," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 1, pp. 51–64, 2018.

[39] S. Tsutsui and P. Collet, *Massively Parallel Evolutionary Computation on GPGPUs*. Springer, 2016.

[40] E. Alba, G. Luque, and S. Nesmachnow, "Parallel metaheuristics: recent advances and new trends," *Intl. Trans. in Operational Research*, vol. 20, no. 1, pp. 1–48, 2013.

[41] D. A. Augusto and H. J. Barbosa, "Accelerated parallel genetic programming tree evaluation with opencl," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 86–100, 2013, metaheuristics on GPUs.

[42] D. Sudholt, *Parallel Evolutionary Algorithms*. Springer Berlin Heidelberg, 2015, pp. 929–959.

[43] A. S. Dufek, D. A. Augusto, H. J. C. Barbosa, and P. L. da Silva Dias, "Multi- and many-threaded heterogeneous parallel grammatical evolution," in *Handbook of Grammatical Evolution*, C. Ryan, M. O'Neill, and J. Collins, Eds. Springer Int. Publishing, 2018, pp. 219–244.

[44] T. Harada and E. Alba, "Parallel genetic algorithms: A useful survey," *ACM Comput. Surv.*, vol. 53, no. 4, aug 2020.

[45] A. S. Dufek, D. A. Augusto, H. J. C. Barbosa, P. L. S. Dias, and J. R. Deslippe, "An efficient fault-tolerant communication algorithm for population-based metaheuristics," in *Proc. of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '21. Association for Computing Machinery, 2021, pp. 1290–1298.

[46] O. A. C. Cortes, M. S. Pais, F. N. Mór, A. Rau-Chaplin, and C. A. M. Marcon, "Differential evolution on a GPGPU: The influence of parameters on speedup and the quality of solutions," in *2015 IEEE Intl. Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 299–306.

[47] A. Mexicano, J. C. Carmona, N. N. Almaza, L. Garcia, and F. Arguelles, "A parallel version of the jade algorithm using gpus," *Digital Signal Processing and Artificial Intelligence for Automatic Learning*, vol. 1, no. 1, 2022.

[48] V. Charilogis and I. G. Tsoulos, "A parallel implementation of the differential evolution method," *Analytics*, vol. 2, no. 1, pp. 17–30, 2023.

[49] M. Bostian, G. Whittaker, A. Sinha, and B. Barnhart, "Incorporating data envelopment analysis solution methods into bilevel multi-objective optimization," in *2015 IEEE Congress on Evolutionary Computation (CEC)*, 2015, pp. 1667–1674.

[50] J.-F. Camacho-Vallejo and C. Garcia-Reyes, "Co-evolutionary algorithms to solve hierarchized steiner tree problems in telecommunication networks," *Applied Soft Computing*, vol. 84, p. 105718, 2019.

[51] T. T. Magalhães, E. Krempser, and H. J. C. Barbosa, "Parallel models of differential evolution for bilevel programming," in *The 20th Intl. Conference on Artificial Intelligence*, 07 2018.

[52] T. T. Magalhães and H. J. C. Barbosa, "Parallel differential evolution algorithms for Stackelberg-Nash bilevel optimization problems," in *2020 IEEE Congress on Evolutionary Computation (CEC)*, 2020, pp. 1–8.

[53] A. Chaabani, S. Bechikh, and L. B. Said, "A co-evolutionary hybrid decomposition-based algorithm for bi-level combinatorial optimization problems," *Soft Computing*, vol. 24, p. 7211–7229, 2020.

[54] A. Chaabani, S. Bechikh, and L. B. Said, "A new co-evolutionary decomposition-based algorithm for bi-level combinatorial optimization," *Applied Intelligence*, vol. 48, p. 2847–2872, 2018.

[55] J. L. Mokhtar Essaid, Lhassane Idoumghar and M. Brévilliers, "Gpu parallelization strategies for metaheuristics: a survey," *Int. J. of Parallel, Emergent and Distributed Systems*, vol. 34, no. 5, pp. 497–522, 2019.

[56] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, p. 341–359, dec 1997.

[57] K. V. Price, *An Introduction to Differential Evolution*. GBR: McGraw-Hill Ltd., UK, 1999, p. 79–108.

[58] S. Das, S. S. Mullick, and P. N. Suganthan, "Recent advances in differential evolution - An updated survey," *Swarm and Evolutionary Computation*, vol. 27, pp. 1–30, 2016.

[59] J. S. Angelo, E. Krempser, and H. J. C. Barbosa, "Differential evolution assisted by a surrogate model for bilevel programming problems," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 1784–1791.

[60] K. A. P. Lagares, J. S. Angelo, H. S. Bernardino, and H. J. C. Barbosa, "A differential evolution algorithm for bilevel problems including linear equality constraints," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, 2016, pp. 1885–1892.

[61] J. S. Angelo, E. Krempser, and H. J. C. Barbosa, "Performance evaluation of local surrogate models in bilevel optimization," in *Machine Learning, Optimization, and Data Science*, G. Nicosia, P. Pardalos, R. Umeton, G. Giuffrida, and V. Sciacca, Eds. Cham: Springer Int. Publishing, 2019, pp. 347–359.

[62] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[63] W. E. Thomson, "A Modified Congruence Method of Generating Pseudo-random Numbers," *The Computer Journal*, vol. 1, no. 2, pp. 83–83, 01 1958.

Biography Section

Amanda S. Dufek is a member of the Application Performance Group at the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory (LBNL) since January 2021. She received her Bachelor (2005) and Master (2008) degrees in Meteorology from the University of São Paulo (USP) located in Brazil, and her Doctor (2015) degree in Computational Modeling from the Brazilian National Laboratory for Scientific Computing (LNCC). Her research interests are: high-performance computing, machine learning, evolutionary algorithms, and atmospheric science problems.

Jaqueline S. Angelo received a B.S. degree in Computer Science from Gama Filho University in 2005 and M.S. and D.Sc. degrees in Computational Modelling from the National Laboratory of Scientific Computing (LNCC, Brazil) in 2008 and 2015, respectively. She is currently a researcher in computational modelling at the Oswaldo Cruz Foundation (Fiocruz, Brazil). Her research interests include linear and nonlinear programming, multi and many-objective evolutionary optimization, metaheuristics, and machine learning.

Douglas A. Augusto is a Research Fellow at the Oswaldo Cruz Foundation (Fiocruz, Brazil). He holds M.Sc. and D.Sc. degrees in Computational Systems from the Federal University of Rio de Janeiro (UFRJ, Brazil). His research focuses on data-driven modeling using machine learning algorithms, high-performance parallel computing, and computational modeling of infectious diseases.

Helio J.C. Barbosa is a retired Senior Technologist from the National Laboratory of Scientific Computing and a retired Professor at the Federal University of Juiz de Fora, Brazil. He holds M.Sc. and D.Sc. degrees in Civil Engineering from the Federal University of Rio de Janeiro and was a visiting scholar at Stanford University from 1988 to 1990. His research interests include evolutionary computation, and he has served as an Associate Editor for the IEEE Transactions on Evolutionary Computation.