DiBB: Distributing Black-Box Optimization

Giuseppe Cuccu
Luca Rolshoven*
Fabien Vorpe*
Philippe Cudré-Mauroux
name.surname@unifr.ch
Exascale Infolab
University of Fribourg
Switzerland

Tobias Glasmachers
tobias.glasmachers@ini.rub.de
Theory of ML Group
Ruhr University Bochum
Germany

ABSTRACT

DiBB (for Distributing Black-Box) is a meta-algorithm and framework that addresses the decades-old scalability issue of Black-Box Optimization (BBO), including Evolutionary Computation. Algorithmically, it does so by creating out-of-the-box a Partially Separable (PS) version of any existing black-box algorithm. This is done by leveraging expert knowledge about the task at hand to define *blocks* of parameters expected to have significant correlation, such as weights entering a same neuron/layer in a neuroevolution application. DiBB distributes the computation to a set of machines without further customization, while still retaining the advanced features of the underlying BBO algorithm, such as scale invariance and step-size adaptation, which are typically lost in recent distributed ES implementations. This is achieved by instantiating a separate instance of the underlying base algorithm for each block, running on a dedicated machine, with DiBB handling communication and constructing complete individuals for evaluation on the original task. DiBB's performance scales *constantly* with the number of parameter-blocks defined, which should allow for unprecedented applications on large clusters. Our reference implementation (Python, on GitHub and PyPI) demonstrates a 5x speed-up on COCO/BBOB using our new PS-CMA-ES. We also showcase a neuroevolution application (11 590 weights) on the PyBullet Walker2D with our new PS-LM-MA-ES.

CCS CONCEPTS

• **Computing methodologies → Randomized search**.

KEYWORDS

Black-Box Optimization, Distributed Algorithms, Parallelization, Evolution Strategies, Neuroevolution

*This paper includes contributions from Rolshoven's and Vorpe's Master Thesis work.

ACM Reference Format:
Giuseppe Cuccu, Luca Rolshoven, Fabien Vorpe, Philippe Cudré-Mauroux, and Tobias Glasmachers. 2022. DiBB: Distributing Black-Box Optimization. In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3512290.3528764

1 INTRODUCTION

Black Box Optimization (BBO) is a broad class of optimization algorithms with the distinguishing advantage that they can be applied, by definition, to *any problem*, independently of the specific application [2]. In principle, this provides a method that is applicable to problems yet unsolved by the current state of the art.

Unfortunately most of these methods have heavy computational requirements. Sophisticated implementations, such as modern Evolution Strategies (ES; Hansen and Ostermeier 18, Wierstra et al. 34), have a high internal computational cost per sample, in exchange for a wide set of properties (see Section 2) that correspond to a higher sample efficiency. Simpler black-box solvers, while less demanding in their performance, conversely suffer from low sample efficiency, which makes them notably *data hungry*.

BBO algorithms in the literature address this trade-off across the whole spectrum. For example, population-based algorithms can evaluate candidate solutions efficiently using embarrassingly parallel computation. More sophisticated implementations however have their computational performance dominated by the update costs, rendering the advantage of parallel population evaluation inconsequential. State-of-the-art algorithms relying on Covariance Matrix Adaptation (CMA; e.g. CMA-ES by Hansen 13) for example have at best *quadratic complexity* [13, 34] in the number of variables for processing a sample, strictly limiting their application (within a sensible time frame) to problems of up to a few thousands variables on today's hardware. Simpler methods meanwhile struggle traversing complex fitness landscapes, requiring disproportionately more samples which also offsets their advantage in practical applications.

As a consequence, while BBO algorithms could in principle be applied to any problem without restriction, their practical application is limited to either trivial problems in high dimensions, or complex problems in very low dimensions.

To fairly evaluate this trade off, this paper distinguishes between *convergence speed*, correspondent to high sample efficiency, and *wall-clock speed*, which effectively measures the practical applicability of a method as the dimensionality and computational complexity grows. In particular we highlight how some algorithms implement an assumption of *separability* between the variables (e.g. sep-CMA-ES [27] and SNES [29]), trading off significant convergence speed for wall-clock speed by relinquishing covariance information altogether [18]. Previous work by the authors also covers a generalization of this trade-off by establishing a *block-diagonal covariance* matrix [6], leveraging the fact that the correlation among variables is not uniform for most complex problems, and that this information is often available to the user based on the target task. This provides initial inspiration for this work, as discussed below.

1.1 Intuition and design

BBO algorithms are designed to work in any unknowable (black-box) environment. By design, BBO algorithms cannot integrate task knowledge, in order to ensure that no illegitimate assumptions are made in their design. In most real-world applications however, some degree of expert knowledge is often available about the end task, making it rather a *gray-box* setting. Expert knowledge about the correlation between variables is often simple to deduce, as it is task-specific.

For example in neuroevolution, where evolutionary algorithms learn the parameters of a neural network, weights of connections entering the same neuron are by necessity highly correlated, as the network's equation aggregates them in a linear combination prior to activation. While weights entering different neurons are not entirely uncorrelated, the expectation on their covariance is (relatively) significantly lower. A similar reasoning is easily made about neurons entering a same layer versus neurons in different layers, and remains true as the network expands towards deep networks [1]. The assumption of *partial correlation*, which allows partitioning the variables into highly intra-correlated **blocks**, leads to the blocks being separable between one another, i.e. low inter-correlation. The corresponding covariance matrix becomes *block-diagonal*, as explored in our previous work [6].

This paper extends the concept by leveraging the underlying assumption of separability between blocks to optimize each block of variables using a different, independent instance of the same BBO algorithm. As a consequence, block-wise computation can now be distributed across multiple machines: not only the individual fitnesses can be evaluated in an embarrassingly parallel fashion, but different blocks can be searched asynchronously, and each BBO instance can be distributed to different nodes in a cluster (see Figure 1).

Based on the above insights, this paper proposes a new meta-algorithm and framework for Distributed Black Box optimization, named **DiBB**. DiBB is particularly suited for large-scale problems evidently structured, as not infrequent (arguably common) in real-world applications. We provide rigorous theoretical arguments for the sample efficiency of this approach in Section 2, and further explore the example application of neuroevolution in Section 4.3.

1.2 Challenges

A major challenge comes with solution evaluation: a BBO instance running on one machine will produce a population of samples that are *incomplete individuals*, as the variables constituting a complete sample are practically scattered across a network. This is addressed in Section 3, by establishing a sparse communication scheme between block instances, and running the fitness evaluation locally on each machine.

[1]The fundamental assumption of partial separability across network layers however does not seem to be very well studied in the literature, despite a considerable body of work on neural network loss landscapes [10, 31]. This can be partially tracked to the limited availability of algorithms making use of partial correlation information.

This paper notably makes **no assumption** or claim on the underlying BBO algorithm of choice. While our experiments explore the applicability of DiBB in the context of modern, sophisticated ESs, it can in principle be applied to any BBO algorithm with minimal interfacing. Even in the extreme case of a known fully-separable problem, a fully-separable BBO [26–29] can still be improved by DiBB as it maintains the separability hypothesis, while immediately generating a *distributable* version of the base BBO, providing constant scaling in wall-clock speed to even the humblest of algorithms.

We analyze the performance of our framework on the classic COCO benchmark, both on the BBOB and BBOB-large-scale suites, using the industry-standard CMA-ES both as a reference and as a base, block-level optimizer in our new DiBB-derived PS-CMA-ES. The sample complexity of our approach typically (as expected) sits in between the full-covariance and the diagonal-covariance versions of CMA-ES, with few notable exceptions where maintaining extra covariance information is actually deceptive/misleading due to problem separability. While algorithms derived by DiBB cannot be in principle superior to the base BBO utilized on low-dimensional problems, using parallel and distributed hardware the new PS-derived algorithm will be considerably (even, arbitrarily, depending on available machines) faster in terms of wall-clock time.

1.3 Contributions

Our key contributions are as follows:

- We provide a novel meta-algorithm, DiBB, that generates a partially-separable version of any available BBO. Our framework also creates a parallel and distributed computation, running on a set of available machines.
- We propose a novel way to further parallelize population-based BBOs such as ESs, going beyond the parallel evaluation of the population's candidate solutions, by running multiple ES instances to optimize approximately separable blocks of variables.
- DiBB preserves the features of the underlying base BBO algorithm through the parallelization, maintaining covariance information at low cost in the specific instances where it provides the largest benefit.
- We derive and introduce two new algorithms to use in our experiments, as Partially Separable versions of CMA-ES and LM-MA-ES respectively: PS-CMA-ES and PS-LM-MA-ES. Creating new algorithms through the DiBB meta-algorithm is straightforward, requires no additional constraints on the base BBO algorithm, and maintains the properties of the latter while enabling distributed computing out of the box.
- We demonstrate experimentally that the wall-clock performance scaling is constant (within practical limits), and that large and deep networks can be trained efficiently on distributed and parallel hardware.

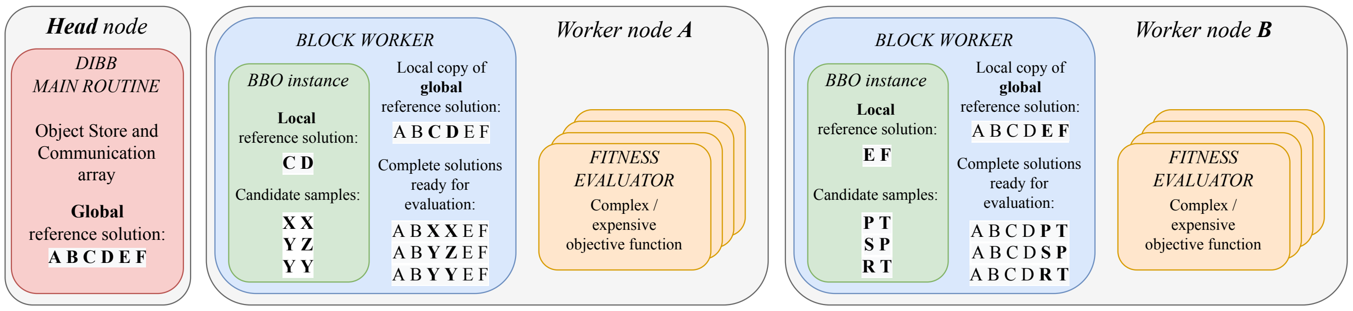
Head node
DIBB MAIN ROUTINE
Object Store and Communication array
Global reference solution:
A B C D E F
BLOCK WORKER
Worker node A
BBO instance
Local reference solution:
C D
Candidate samples:
X X
Y Z
Y Y
Local copy of global reference solution:
A B C D E F
Complete solutions ready for evaluation:
A B X X E F
A B Y Z E F
A B Y Y E F
FITNESS EVALUATOR
Complex / expensive objective function
BLOCK WORKER
Worker node B
BBO instance
Local reference solution:
E F
Candidate samples:
P T
S P
R T
Local copy of global reference solution:
A B C D E F
Complete solutions ready for evaluation:
A B C D P T
A B C D S P
A B C D R T
FITNESS EVALUATOR
Complex / expensive objective function

Figure 1: Example architecture of DiBB as a framework. This is a sample instantiation on a cluster with 3 nodes: one acts as the head node, running the main routine, while two worker nodes encapsulate the underlying BBO and host the pools of Fitness Evaluators. This schema highlights how DiBB leverages Partial Separability in the construction of full samples ready for evaluation, and how the (potentially expensive) objective function is evaluated locally on the worker nodes.

2 A PRIMER ON EVOLUTION STRATEGIES

DiBB can be applied to any BBO algorithm to great and immediate advantage: even with methods that already treat the parameters as separable, DiBB provides a parallel and distributed implementation at the only cost of minimal overhead (for nontrivial applications). In this section however, we highlight modern Evolution Strategies as the state-of-the-art family of continuous black-box optimization methods, which we expect to gain the most from this new approach. The rationale here is that these methods have proven and popularized the importance of maintaining covariance information for search performance, and are thus best positioned to gain from DiBB's scalability into distributed computation. This choice however should not be interpreted as limited to neuroevolution applications, as DiBB users can adapt the partitioning of variables into blocks depending on each problem at hand.

ES are direct search methods, which optimize a black-box objective function $f : \mathbb{R}^d \to \mathbb{R}$ by sampling candidate points from an adaptive Gaussian distribution. We briefly review the types of ES most relevant for our discussion. The classic variant is the (1+1)-ES [26]. Its central algorithmic mechanism is step-size adaptation, i.e. its ability to actively adapt the standard deviation $\sigma > 0$ of its Gaussian sampling distribution $\mathcal{N}(m, \sigma^2 I)$ to the current needs. For the last 20 years, CMA-ES [18] has been the gold standard in ES research. Many variants exist, such as Natural Evolution Strategies (NES; Wierstra et al. 34). Its most important mechanism going beyond "simple" step-size adaptive ES is covariance matrix adaptation (CMA), which means that not only the global step size σ, but also the full covariance matrix C of the Gaussian $\mathcal{N}(m, \sigma^2 C)$ is adapted to the problem at hand.

CMA-ES is a powerful optimizer; however, it was not designed for high-dimensional applications with hundreds of thousands of variables or more. Its internal parameters are not tuned with such a regime in mind, and learning a full covariance matrix with $\frac{d(d+1)}{2}$ parameters is inherently slow. Such problems are commonly addressed by placing specific restrictions on C, such as being represented by a diagonal matrix [27, 29], or a diagonal plus a low-rank matrix [1, 22, 23], or a block-diagonal matrix [6]. The number of parameters of the covariance matrix can hence be chosen flexibly in the range d to $\frac{d(d+1)}{2}$, allowing scaling to higher dimensional problems by trading off covariance awareness.

2.1 ES for Neuroevolution

The application of ES to machine learning problems and to RL in particular has a multi-decade history [19, 20]. In 2017, the work of Salimans et al. [28] sparked a renewed interest in ES by showcasing how to successfully exploit the embarrassingly parallel nature of individual objective function evaluations in populations-based algorithms, proposing a considerable speed-up in the learning process. This triggered a large body of work on neuroevolution based on ES over the past few years, see e.g. Chrabaszcz et al. [4], Cuccu et al. [7], Ha and Schmidhuber [12], Plappert et al. [25], Stanley et al. [32] and references therein.

2.2 Convergence Rates and Computational Complexity

Due to Taylor's theorem, local optima of d-dimensional C^2 functions are well approximated (up to $O(\|x - x^*\|^3)$) by convex quadratic functions $f(x) = \frac{1}{2}(x - x^*)^T H(x - x^*)$. The computational complexity of solving this problem with an ES to a fixed target precision $\varepsilon > 0$ is of the form $O(d \cdot \kappa(H) \cdot \log(1/\varepsilon))$, where $\kappa(H)$ denotes the condition number of the Hessian H [15, 21]. Hence, a step-size adaptive ES achieves linear convergence with rate $O\big(1/(d \cdot \kappa(H))\big)$.

The linear dependency on d is optimal for comparison-based optimization [11], but the dependency on H is sub-optimal. The advantage of maintaining covariance information is that the factor $\kappa(H)$ is improved to $\kappa(HC^*)$, where C^* denotes the optimal covariance matrix available to the ES. When approaching $C^* = H^{-1}$, methods maintaining full covariance achieve the optimal value $\kappa(HC^*) = 1$. In effect, as expected from a pseudo second order method, the convergence rate is *independent* of H. A diagonal covariance matrix C^* acts as a diagonal pre-conditioner, with varying effectiveness depending on the problem. Obviously, block-diagonal [6] and low-rank [22, 23] covariance matrix representations are in-between. The block-diagonal case notably generalizes both the full-covariance and diagonal-covariance implementations when the size of the blocks is d and 1 respectively (corresponding to 1 block and d blocks in turn).

The OpenAI-ES [28], while being highly distributable, features neither step-size adaptation nor covariance adaptation. Based on the NES framework of Wierstra et al. [34], it leverages the ability of ES to estimate the natural gradient of f from samples, and then applies the ADAM optimizer on top. In effect, this is roughly comparable to using a diagonal C.

CMA has a price in terms of *algorithm internal* complexity, and in addition the adaptation process is slow even in terms of sample complexity—due to large hidden constants in its performance estimation. The above convergence rates measure time in terms of the number of objective function evaluations (sample complexity). When scaling up CMA to high dimensions however, we need to take the following concepts into consideration. Algorithm internal complexity refers to the required (amortized) number of operations needed for creating a sample and for updating the internal state—the covariance matrix in particular. This concept will be re-explored later when introducing distributed computing, as DiBB employs multiple machines to evaluate proportionally more samples without impacting wall-clock time. For now, regarding sample complexity, we distinguish between the number of samples needed to learn the covariance matrix, and the number of samples needed to solve the problem.

Learning C with up to $\Theta(d^2)$ parameters is sample-inefficient. For example, a (small) neural network with $d = 10^4$ weights results in $\frac{d(d+1)}{2} \approx 5 \cdot 10^7$ parameters of the covariance matrix. Learning these takes hundreds of millions of samples, each of which can be significantly expensive: imagine for example a continuous control task involving a robot interacting in a physics simulation. This quickly becomes far too slow for practical use, even without taking into consideration the (typically expensive) covariance update step in the underlying algorithm. For larger d, even the storage of the full covariance matrix C quickly becomes prohibitive. Furthermore, performing computations with C scales at least linear with the number of its parameters, which amounts to an internal complexity of $\Omega(d^2)$ for full CMA. Since network evaluation scales linearly with the number of weights d (in a direct encoding scheme), CMA quickly becomes the computational bottleneck. Therefore, a different trade-off between fast convergence and internal complexity is needed, which can be realized for example with block-diagonal and low-rank structures, and combinations thereof. Or, in the case of the proposed work, by leveraging the partial correlation assumption to construct a block-diagonal covariance matrix.

2.3 Implications for Neural Network Training

Diagonal, block-diagonal, and low-rank schemes successfully lower both internal and sample costs of CMA significantly (at the expense of higher sample complexity for solving the overall problem). Therefore, they are key to the application of modern ES based on CMA to real-world problems, especially involving expensive physics simulations as common in reinforcement learning applications.

In neural network training, weight spaces are often extremely high-dimensional. However, they also come with a canonical structure, induced by the network topology. Hence, a block-diagonal covariance structure with one block per layer (or per neuron) is a natural choice. It should be noted that there exist approaches for identifying a problem decomposition automatically [24], if needed, extending the applicability of DiBB to problems where problem structure and variable correlation are not initially known to the user.

If H has a block structure (f is separable), then *sequentially* optimizing all blocks in isolation is as fast as optimizing the full problem. This is a direct consequence of the $O(d)$ scaling of the sample complexity discussed above, and leads to the following insight:

Solving b independent sub-problems with $k = d/b$ variables each *in parallel* results in a b-fold speed-up over solving the full problem with d variables. Importantly, since b and hence k is user-defined, and provided that block remains of constant size while scaling the dimension, this allows the *constant runtime* (depending on communication overhead, thus implementation dependent) that we see in DiBB's experimental results.

It is understood that this comes at a cost if the separability assumption is violated. However, the block structure offers a further benefit:

Solving b independent sub-problems with $k = d/b$ variables each with an ES featuring CMA results in $O(k^2)$ sample and internal complexity for covariance learning, in contrast to $O(d^2)$ for full CMA. If the number of blocks b scales linearly with the problem size d, or $k \in O(1)$, then CMA becomes feasible for arbitrarily large problems.

These two insights offer a novel route towards highly parallel and at the same time more sample-efficient neuroevolution strategies. In addition to the embarrassingly parallel evaluation of a population of candidate points, multiple blocks can be optimized in parallel. Given enough cores, higher parallelism results in a b-fold speed-up. As an additional benefit, CMA can be applied within each block of variables. Provided that the problem has an (approximately) separable structure, CMA results in improved sample efficiency, yielding a further speed-up.

3 METHOD

The DiBB framework is inspired by our previous work with BD-NES [6] in the sense that we utilize a block-diagonal covariance matrix and partial separability towards distributing the block computation across dedicated machines. The original design however was specifically tailored to the NES family of ES, while the meta-algorithm facet of DiBB makes it applicable not only to any ES but to any BBO algorithm without restrictions, providing at the same time a parallel and distributed implementation with limited overhead.

Here is a short summary of how to use DiBB:

(1) The user defines a partition of the parameters, typically based on expert knowledge of the application. For example in neuroevolution, consider weights of connections entering a same neuron or layer.
(2) The main method, launched on a *head node*, spawns the run control routine plus the object store that maintains shared data and handles communication. This follows modern best practices in distributed processing [8].
(3) The control routine then launches one *Block Worker* (BW) for each parameter block, one on each of the available machines.

The BW encapsulates the actual BBO algorithm, and runs fully asynchronously from the others, though contemporary. The BWs exchange information by uploading to the head node the state of their search after each update (i.e. a *generation* in ES); updates from the other nodes are also pulled before generating a new population.

(4) In our implementation, each BW can spawn a pool of *Fitness Evaluators* (FE) on the same machine, used to automatically manage limited computational resources (such as available CPUs).
(5) Alternatively, the user can request to evaluate trivial optimization functions on the BW directly (either sequentially or using multithreading), which is useful when the overhead of maintaining a discrete pool of evaluators would be significant with respect to the cost of the actual evaluation task.

The Block Workers communicate with the head node in each generation, while generation cycles are defined asynchronously and autonomously by each BW. Consider a problem with d variables $x_1, \dots, x_d$, and a BW optimizing b variables $x_a, \dots, x_{a+b-1}$, denoted as the vector x_B for short. The head node maintains a *reference solution* $\bar{x} \in \mathbb{R}^d$, which fulfills a two-fold purpose: it serves as an anytime-estimate of the state of the search (current optimum), and it provides a unifying context to the BWs and the FEs.

Intuitively, each BW is only aware of the variables in one block, and can only generate samples for the corresponding variables. These incomplete samples however need to be constructed as part of a complete solution in order to be scored on the task. In our recurring example of neuroevolution, the sample could correspond to the weights for a neuron, or layer, while obviously only full networks can be evaluated on the task. We address this issue by leveraging once again our assumption of partial separability. After our hypothesis of the correlation across blocks being negligible, we can evaluate each block in isolation by inserting it in the context of the (current) reference solution, as obtained from the head node.

Hypothesize for a moment that there is no correlation inter-block—we will address the validity of this statement just below. In this case, each layer can be scored fairly by constructing a full reference network, then swapping the corresponding weights in the target layer for the block sample, and finally evaluating the resulting complete network on the task. Different independent samples from a same block (individuals) would receive fair evaluation in this fashion as long as they are evaluated on the same reference network. This is in fact constructed by assembling the partial sample into the global reference solutions. Partial samples are constructed by aggregating the reference or center sample from each of the BBO instances running on each block, which as mentioned is maintained in the head node by each block instance at the end of each generation.

More formally: at the start of each generation, the BW receives the current reference solution $\bar{x}_1, \dots, \bar{x}_d$ from the head node.[2] The block-level ES samples a population of candidate solutions $y^1, \dots y^\lambda \in \mathbb{R}^b$. If b is small, then sampling from a Gaussian with full covariance matrix $\mathcal{N}(m, \sigma^2 C)$ is feasible. The b-dimensional points are injected into the reference solution by constructing the d-dimensional vectors $x^1, \dots x^\lambda \in \mathbb{R}^d$ according to the following rule:

$$x_i^j = \begin{cases} y_{i-a}^j & \text{if } a \le i \le a+b \\ \bar{x}_i & \text{otherwise} \end{cases}$$

The vectors $x^1, \dots, x^\lambda$ are passed to the (thread or node) pool of Fitness Evaluators for fitness evaluation. Once all are computed, the ES updates its internal state based on the initially produced partial candidates and the fitness values obtained from the full-individual evaluations. This includes updating the sampling mean m and optionally further parameters like step size, evolution paths, and covariance matrix, depending on the base BBO and with no tampering from DiBB itself. Finally, it sends the updated mean m back to the head node, which incorporates it into its reference solution by overwriting x_B with m. This makes the update available for the next cluster node preparing for its next generation.

The last issue remaining is that of course we cannot expect the weights entering different neurons or layers to be entirely separable; after all, they do belong to the same network and they are thereby all contributing to the final output in its equation. Since all blocks are searched at the same time, this induces a problem of *moving target*, where the score of a block sample depends on the global state of the search (in the form of the current whole reference solution, as held by the head node), which changes constantly as every BBO instance asynchronously sends an update to the global reference state.

Empirically we verify that the impact on the algorithm is however only minimal: after all, our hypothesis is much stricter than in fully-separable implementations. While in principle and in theory full-covariance algorithms have better sample efficiency, their hypothesis is for all parameters to be meaningfully correlated. In real-world scenarios however it is most common to have complex applications with high dimensionality and relatively sparse or low covariance. A full-covariance algorithm still needs to learn the full covariance matrix each time, at significant computational costs, which effectively lowers their sample efficiency. DiBB on the other hand allows for integrating expert knowledge in its blocks construction, sidestepping the problem of learning covariance information between blocks that can be considered independent. As a result and in practice, we have seen no measurable advantage or disadvantage on either approach from this perspective.

The proposed setup directly reflects the added level of parallelism, compared to a standard ES. Traditional implementations usually restrict parallelism to maintaining a pool of evaluators to speed up the evaluation of independent samples using multiple cores. In our setup, a second level of parallelism is established in terms of the Block Workers, which operate independently except for sparse communication with the head node—itself entirely dedicated to the task to ensure high responsiveness.

Per generation and BW, d real values need to be communicated to and from the head node, to update the global reference solution and to obtain a local copy of it respectively. For a constant block size and a number of nodes linear in d, the total network traffic per generation grows quadratically with d. For extremely large problems, this may eventually limit the effectiveness of the head node.

[2] It would suffice to send $\bar{x}_1, \dots, \bar{x}_{a-1}, \bar{x}_{a+b}, \dots, \bar{x}_d$ to the BW, but the difference has been negligible in our experiments so far up to 30 blocks.

The problem is smartly sidestepped by Salimans et al. [28] by synchronizing the *entropy* (random number generation) across the machines, which makes them generate the same samples, however at the price of a fully-synchronous implementation (and greatly simplified algorithm). The same method can be implemented into DiBB to allow a Block Worker to spawn Fitness Evaluators across multiple machines, which would immediately improve performance particularly in the case of large population sizes. With DiBB however, the actual updates would remain asynchronous, which implies that the FEs spawned in the cluster could actually be shared between BWs. Since this communication overhead occurs only once per (block-wise) generation in DiBB, and not once per fitness evaluation, our implementation has considerable less pronounced network overhead than in the distribution of fitness evaluation in a plain ES.

4 EXPERIMENTS

This section describes the setup used to empirically assess the performance of DiBB. With our experiments, we aim to address the following research questions:

- Q1: How well does the block-diagonal approach work, compared to diagonal- and full-covariance CMA?
- Q2: How does DiBB's performance scale to a large number of machines and cores?
- Q3: Is DiBB well-suited for neuroevolution applications?

We assess the first two questions on the standard COmparing Continuous Optimizers (COCO) Black Box Optimization Benchmark (BBOB), using both the standard [17] and the *large-scale* [9] benchmark suites. For answering the third question, we showcase a neuroevolution application in the challenging OpenAI Gym 2D Walker environment.

4.1 Setup and reference implementation

We tested DiBB on a variety of hardware solutions. The COCO-BBOB experiments were run on a cluster of 24 low-performance machines, all based on an Intel$^{(R)}$ Core$^{(TM)}$ i7-2600 CPU @ 3.40GHz (4 cores/8 threads each), and 32 GB of RAM. These machines are far from state-of-the-art performance, which leaves a significant margin of improvement for the timings presented in our results. This decision was taken to encourage interested labs to reproduce our results on whatever hardware they can put together, without expectation of dedicating any significant budget.

The cluster setup is simplified by the included managing scripts. Running on a single machine with a single block and no Fitness Evaluators roughly corresponds to running the underlying BBO algorithm alone (plus overhead, and with parallel fitness evaluation), and can be achieved without setup with a syntax alike to CMA-ES. Spawning multiple BWs and FEs automatically scales to the available resources, as declared in the managing script using a simple list of network IPs.

The experiments below are based on our reference implementation of DiBB written in Python, which leverages the Ray distributed computation library[3]. The code is released open source on GitHub both for the experiment design[4] and framework implementation[5], and collaborations and contributions are encouraged. DiBB is also available through PyPI[6] for ease of adoption, using the standard `pip` installer.

[3]https://www.ray.io/ — a Python framework for distributed computing

Number of dimensions	Number of blocks	Block size	Duration
80	16	5	02h 47m 53s
160	16	10	04h 11m 15s
320	16	20	07h 01m 38s
640	16	40	12h 16m 02s

Table 1: Timing results for Experiment 3: BBOB large-scale suite with a fixed number of blocks. Larger problem dimensions are addressed here with larger block sizes, which leads to more correlation information being considered in each block. The duration of the experiments increases linearly with the block size for small dimension increments.

4.2 COCO BBOB

COCO provides multiple suites covering a broad range of test cases. The experiments presented in this work are based on the BBOB *standard large-scale* suites. The BBOB standard suite comes with 24 noise-free real-parameter single-objective benchmark functions in dimensions $d \in \{2, 3, 5, 10, 20, 40\}$ [17]. All of the functions have their global optimum in $[-5, 5]^d$ by design, where d is the dimensionality of the problem. The functions are divided into five groups: separable functions (f1-f5), moderately conditioned functions (f6-f9), ill-conditioned functions (f10-f14), multi-modal functions (f15-f19), and weakly structured multi-modal functions (f20-f24). We expect the performance of DiBB to vary accordingly to the actual validity of our assumption of partial separability.

The BBOB large-scale suite contains the same 24 same functions which are found in the standard suite, but scales up the available number of dimensions to $d \in \{20, 40, 80, 160, 320, 640\}$. This suite however introduces heuristics to decrease the computational cost of a selection of functions in a large-scale setting. We refer the interested reader to Elhara et al. [9] for further details.

The BBOB suite provides a broad range of problems representative of many use-cases in the spectrum of Black-Box Optimization. Particularly, and by design, several edge cases are present that are unlikely to be encountered in real applications but provide compelling insights on the performance and applicability of the tested algorithm. BBOB problems are specifically designed to be solved to high precision with the goal to test for scale invariance; therefore d is not in the thousands, without the lower dimensionality significantly impacting the problem complexity. Scalability of d, and the distinction between separable and non-separable problem classes, allows us to systematically evaluate the effect of DiBB's block structure on its performance. COCO greatly facilitates this

[4]Experiments code to reproduce our BBOB/COCO results: https://github.com/eXascaleInfolab/dibb_coco
[5]DiBB reference implementation: https://github.com/giuse/dibb/
[6]Python Package Index (`pip install dibb`): https://pypi.org/project/dibb/

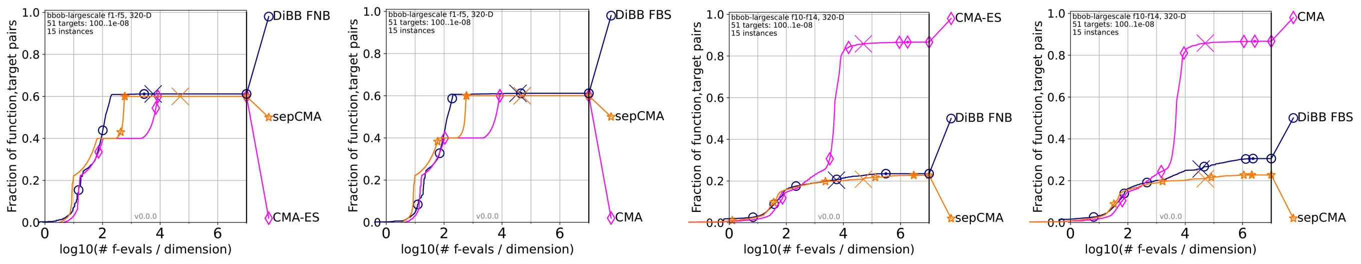

Figure 2: ECDF plots of the performance of PS-CMA-ES, sep-CMA-ES and standard CMA-ES in dimension 320. From left to right: f1-f5 with Exp. 1/Exp. 3, Exp. 2/Exp. 4, then f10-f14 with Exp. 1/Exp. 3, Exp. 2/Exp. 4. The curves show the fraction of precision targets reached over time, measured as number of function evaluations divided by problem dimension, on a logarithmic scale. For additional information on ECDF plots, please refer to [16].

Number of dimensions	Number of blocks	Block size	Duration
40	1	40	25h 13m 47s
80	2	40	43h 26m 57s
160	4	40	40h 21m 01s
320	8	40	44h 53m 29s
640	16	40	61h 20m 11s

Table 2: Timing results for Experiment 4: BBOB large-scale suite with fixed block size (5× budget). Larger problem dimensions are addressed here with increasing the number of blocks. Since each block is optimized in parallel, the increase in run time of the experiments is only limited to communication overhead, thus small as problem dimensions grow.

process by providing publicly available performance data for many state-of-the-art algorithms.[7]

We test the scaling of DiBB's performance as we vary in turn the problem dimensions and the block size, from the related but radically different perspectives of sample efficiency (i.e. convergence speed, how many samples it takes for the algorithm to reach convergence) and effective run time (i.e. wall-clock speed, how long it actually takes for the algorithm to converge in real time). In the case of DiBB, run time is significantly (positively) impacted by its inherent distributed implementation, as each BW evaluates the fitness of its individuals locally in its dedicated machine, asynchronously from all other BWs.

To this end, we ran the following experiments:

(1) Constant number of blocks, increasing d and block size, on the BBOB suite (`bbob_fnb`)
(2) Constant block size, increasing d and number of blocks on the BBOB suite (`bbob_fbs`)
(3) Constant number of blocks, increasing d and block size on the large-scale suite (`bbob_ls_fnb`)
(4) Constant block size, increasing d and number of blocks on the large-scale suite (`bbob_ls_fbs`)
(5) Comparison of the wall-clock speed between plain CMA-ES and PS-CMA-ES using different block sizes

Complete details of the experimental setup for reproducibility purpose are found in Appendix A.1.

[7] See https://numbbo.github.io/data-archive/bbob/.

We chose to focus on the group of separable (f1-f5) and ill-conditioned functions (f10-f14). The first group contains problems that can be solved easily by DiBB, while the second group is extremely hard since it strongly violates the separability assumption.

Prototypical results of the COCO/BBOB experiments are found in Figure 2 (full results in Appendix A.1). The Empirical Cumulative Distribution Function (ECDF) plots [17] show the fraction of reached (precision) targets over dimension-normalized time on a log-scale, so "higher is better".

Sample Efficiency. On fully separable problems f1-f5, all algorithms perform roughly the same in terms of sample complexity. In other words, covariance terms can be dropped at no cost, and blocks can be optimized independently. This confirms the theoretical predictions from Section 2. Interestingly, in some cases PS-CMA-ES outperforms standard CMA-ES (if only slightly). We expect such a behavior in this particular (artificial and extreme) case, since effectively dedicating resources to learn the (initially misleading and then uninformative) off-diagonal covariance terms can be detrimental.

For the ill-conditioned non-separable problems f10-f14, the performance of PS-CMA-ES is close to sep-CMA-ES and far worse than standard CMA-ES, again as expected since the unmodeled covariance terms (with unrealistic correlations, extremely close to ±1) dominate performance. Yet, unsurprisingly, larger blocks immediately result in better sample complexity. This finding confirms the theoretical prediction that the assumption of partial separability is crucial, and that the user can find an ideal trade-off by optimizing the block hyperparameters.

Taken together, the two results imply that DiBB-derived algorithms applied to a problem with block structure greatly outperform an ES with diagonal covariance matrix by simply being more sample-efficient within each block, while full CMA offers no further advantage. This answers question Q1.

Timings. Most experiments were run on the low-performance cluster described above. For three of the experiments however we tested the flexibility of DiBB by leveraging a new cluster of only three nodes but with Intel(R) Xeon(R) CPU E5-2620 v4 processors, with 16 cores (32 threads) @ 2.10GHz and 128 GB RAM. These were used for the 5d BBOB suite with one block (thus running on a single machine), the 40d BBOB suite with two blocks (running on three machines: one head node and two for the Block Workers), and the

Blocks	Duration	Relative Time	Speed Gain
1	17h 43m 41s	100.00%	0.00%
2	17h 33m 30s	99.04%	0.97%
4	06h 32m 19s	36.88%	171.13%
8	03h 56m 56s	22.27%	348.94%
16	03h 06m 05s	17.49%	471.62%

Table 3: Timing results for Experiment 5: BBOB large-scale suite in 160 dimensions using different block sizes. The first row (one block) corresponds to running CMA-ES without DiBB. When using only two blocks, we observe almost the same wall-clock time. For larger number of blocks however, the time is significantly reduced.

40d BBOB large-scale suite with one block (another single-machine run).

Tables 1 and 2 compare the run times of DiBB with different numbers of blocks, block sizes, and dimensions on the BBOB large-scale suite. Several effects come together: in larger dimensions, function evaluation time and communication overhead grow linearly. Due to the fixed budget multiplier used in COCO, the overall function evaluation budget also grows linearly. On the other hand, CMA's computational effort grows quadratically in the block size. We clearly observe that the runtime grows far more benign when using a fixed block size, which indicates that CMA overhead indeed quickly becomes the dominating term. Hence, the block size should be kept tightly under control. This answers question Q2. Additionally, Table 3 compares the wall-clock speed of PS-CMA-ES using different block sizes on the 160d BBOB large-scale problems suite. For one vs. two blocks, there is almost no difference in the duration of the experiment. However, for four or more blocks, the wall-clock time diminished drastically, leading to a speed gain of 471%. Note that for BBOB and most other benchmark problems, evaluations are unrealistically cheap, so DiBB's parallelization overhead becomes relatively significant. For a more realistic function evaluation cost (i.e. as little as 10ms) the overhead becomes negligible, highlighting the benefit of parallel evaluations.

4.3 PyBullet Walker 2D

Although DiBB can be applied to any existing BBO and problem, its value in neuroevolution applications has undeniably been one of the original inspirations, particularly in relation to neural networks of large size. Therefore we present our results on a complex reinforcement learning control task: the Walker 2D environment from PyBullet [5], instantiated through the OpenAI Gym [3].

In this task, an agent controls a basic 2D robotic walker using the PyBullet physics simulator. The observation is a 22-dimensional reading of the environment (e.g. aperture and angular velocity of each joint, etc.), while the action is a 6-dimensional torque-control signal. The goal of the task is for the robot to walk the farthest distance possible in the allotted time, without toppling, which in turn requires learning a usable gait.

The policy network is feed-forward and fully connected, composed by two hidden layers of sizes $[128, 64]$, using ReLU activations. The output layer is composed of six neurons with rescaled tanh activation normalized to the range of motor commands.

The network, totaling 11 590 weights and 198 neurons, is trained with DiBB applied to LM-MA-ES [23] to derive PS-LM-MA-ES, running four blocks on four machines. The first and last block corresponded to the input layer (2 944 parameters) and output layer (390 parameters) were run directly on separate machines. As part of the testing, the output layer block was run on one of the 16-cores machine, together with the main routine (rather than a dedicated head node), to no noticeable difference in performance. The connection between the two hidden layers however (accounting for 8 256 weights) was instead split in two blocks, as the connections entering two groups of 32 neurons each (4 128 weights), simply to further test DiBB's flexibility and substantially reduce run-time. This setup achieved a score of **1 126** (average over 100 runs, episode length capped at 1 000 frames) in 25 hours. As a baseline we used Random Weight Guessing [30], which reached a score of 42 within 1 000 trials. To the authors' knowledge, these are the first results on this benchmark that use neuroevolution. This answers question Q3.

5 CONCLUSIONS

This paper introduces DiBB, a meta-algorithm and framework that addresses the scalability and performance issues of black-box optimization (notably including Evolutionary Computation), by constructing a Partially-Separable (PS) version of the underlying BBO algorithm of choice, and then parallelizing and distributing its computation across a number of available machines. Configuring new BBO algorithms or adding and removing machines takes *literally minutes* in our reference implementation, with no limitation or requirement on the underlying BBO algorithm nor on the problem.

This allows DiBB to e.g. scale state-of-the-art Evolution Strategies to large-dimensional problems while maintaining key advanced features such as scale invariance and adaptable step-size. This is a significant step forward over recent large-scale implementations, which renounced such advanced features as a price for their flavor of distributed computation. DiBB instead runs multiple instances of the base algorithm each on a subset of parameters selected for being highly intra-correlated, each on a separate machine.

The resulting performance scales *constantly* with the number of machines available, as the overall computational complexity is bound not in the total number of variables, but in the (arbitrary, user-defined) size of the largest block. The algorithm complexity scales constantly with the number of blocks as long as more machines are available, but for a limited communication overhead. Our results on scaling CMA-ES to large dimensions (via our new PS-CMA-ES) on the COCO BBOB large-scale suite reaches an unprecedented number of dimensions simply by adding more (but cheap, old, low-performance) machines, for a significant speed-up.

We also included a neuroevolution demonstration, training a network of 11 590 weights using PS-LM-MA-ES in 25 hours on four low-performance machines, learning a complex robotic task (2D Walker) simulated in PyBullet. Our code is open source and available on GitHub, and includes out-of-the-box both PS-CMA-ES and PS-LM-MA-ES.

Future work includes exploring the dynamics of different BBO algorithms, and scaling to larger models—potentially even non-differentiable, as smoothness is not a requirement for BBO.

REFERENCES

[1] Youhei Akimoto, Anne Auger, and Nikolaus Hansen. 2014. Comparison-based natural gradient optimization in high dimension. In *Genetic and Evolutionary Computation Conference*. ACM, 373–380.

[2] Charles Audet and Warren Hare. 2017. *Derivative-free and blackbox optimization*. Springer.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:arXiv:1606.01540

[4] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. 2018. *Back to basics: Benchmarking Canonical Evolution Strategies for Playing ATARI*. Technical Report 1802.08842. arXiv.org.

[5] Erwin Coumans and Yunfei Bai. 2016–2021. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

[6] Giuseppe Cuccu and Faustino Gomez. 2012. Block diagonal natural evolution strategies. In *International Conference on Parallel Problem Solving from Nature*. Springer, 488–497.

[7] Giuseppe Cuccu, Julian Togelius, and Philippe Cudré-Mauroux. 2019. Playing Atari with six neurons. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 998–1006. https://exascale.info/assets/pdf/cuccu2019aamas.pdf

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113. https://doi.org/10.1145/1327452.1327492

[9] Ouassim Ait Elhara, Konstantinos Varelas, Duc Manh Nguyen, Tea Tušar, Dimo Brockhoff, Nikolaus Hansen, and Anne Auger. 2019. COCO: The Large Scale Black-Box Optimization Benchmarking (bbob-largescale) Test Suite. *arXiv preprint arXiv:1903.06396* (2019).

[10] Stanislav Fort and Stanislaw Jastrzebski. 2019. Large scale structure of neural network loss landscapes. *Advances in Neural Information Processing Systems* 32 (2019), 6709–6717.

[11] Hervé Fournier and Olivier Teytaud. 2011. Lower bounds for comparison based evolution strategies using vc-dimension and sign patterns. *Algorithmica* 59, 3 (2011), 387–408.

[12] David Ha and Jürgen Schmidhuber. 2018. *Recurrent world models facilitate policy evolution*. Technical Report 1809.01999. arXiv.org.

[13] Nikolaus Hansen. 1996. Adapting Arbitrary Normal Mutation Distributions in Evolution Strategies: The Covariance Matrix Adaptation. In *IEEE International Conference on Evolutionary Computation, 1996*. 312–317.

[14] Nikolaus Hansen. 2019. A global surrogate assisted CMA-ES. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Prague Czech Republic, 664–672. https://doi.org/10.1145/3321707.3321842

[15] Nikolaus Hansen, Dirk V Arnold, and Anne Auger. 2015. Evolution strategies. In *Springer handbook of computational intelligence*. Springer, 871–898.

[16] Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Pošík. 2010. Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*. 1689–1696.

[17] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions*. Technical Report RR-6869. INRIA.

[18] Nikolaus Hansen and Andreas Ostermeier. 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation* 9, 2 (2001), 159–195.

[19] Verena Heidrich-Meisner and Christian Igel. 2009. Neuroevolution strategies for episodic reinforcement learning. *Journal of Algorithms* 64, 4 (2009), 152–168.

[20] Christian Igel. 2003. Neuroevolution for Reinforcement Learning using Evolution Strategies. In *The 2003 Congress on Evolutionary Computation (CEC'03)*, Vol. 4. IEEE, 2588–2595.

[21] Jens Jägersküpper. 2006. How the (1+1)-ES using isotropic mutations minimizes positive definite quadratic forms. *Theoretical Computer Science* 361, 1 (2006), 38–56.

[22] Ilya Loshchilov. 2014. A Computationally Efficient Limited Memory CMA-ES for Large Scale Optimization. In *Genetic and Evolutionary Computation Conference*. ACM, 397–404.

[23] Ilya Loshchilov, Tobias Glasmachers, and Hans-Georg Beyer. 2018. Large Scale Black-box Optimization by Limited-Memory Matrix Adaptation. *IEEE Transactions on Evolutionary Computation* 99 (2018).

[24] Mohammad Nabi Omidvar, Xiaodong Li, Yi Mei, and Xin Yao. 2013. Cooperative co-evolution with differential grouping for large scale optimization. *IEEE Transactions on Evolutionary Computation* 18, 3 (2013), 378–393.

[25] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. 2017. *Parameter space noise for exploration*. Technical Report 1706.01905. arXiv.org.

[26] Ingo Rechenberg. 1973. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog.

[27] Raymond Ros and Nikolaus Hansen. 2008. A Simple Modification in CMA-ES Achieving Linear Time and Space Complexity. In *Parallel Problem Solving from Nature – PPSN X*, Günter Rudolph, Thomas Jansen, Nicola Beume, Simon Lucas, and Carlo Poloni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 296–305.

[28] Tim Salimans, Jonathan Ho, Xi. Chen, and Ilya Sutskever. 2017. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. Technical Report arXiv:1703.03864. arxiv.org.

[29] Tom Schaul, Tobias Glasmachers, and Jürgen Schmidhuber. 2011. High Dimensions and Heavy Tails for Natural Evolution Strategies. In *Proceedings of the 13th annual conference on Genetic and Evolutionary Computation (GECCO)*. 845–852.

[30] Jürgen Schmidhuber, S Hochreiter, and Y Bengio. 2001. Evaluating benchmark problems by random guessing. *A Field Guide to Dynamical Recurrent Networks, ed. J. Kolen and S. Cremer* (2001), 231–235.

[31] Mahdi Soltanolkotabi, Adel Javanmard, and Jason D Lee. 2018. Theoretical insights into the optimization landscape of over-parameterized shallow neural networks. *IEEE Transactions on Information Theory* 65, 2 (2018), 742–769.

[32] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. 2019. Designing neural networks through neuroevolution. *Nature Machine Intelligence* 1, 1 (2019), 24–35.

[33] Konstantinos Varelas. 2019. Benchmarking Large Scale Variants of CMA-ES and L-BFGS-B on the Bbob-Largescale Testbed. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1937–1945. https://doi.org/10.1145/3319619.3326893

[34] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber. 2014. Natural Evolution Strategies. *Journal of Machine Learning Research* 15 (2014), 949–980.