Massively parallel CMA-ES with increasing population

David Redon, Pierre Fortin, Bilel Derbel, Miwako Tsuji, Mitsuhisa Sato

Massively parallel CMA-ES with increasing population

David Redon[1], Pierre Fortin[2], Bilel Derbel[1], Miwako Tsuji[3], and Mitsuhisa Sato[3]

[1]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2]Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[3]RIKEN Center for Computational Science, Kobe, Hyogo, Japan
Emails: {david.redon, pierre.fortin, bilel.derbel} @univ-lille.fr
{miwako.tsuji, msato} @riken.jp

November 13, 2025

Abstract

The Increasing Population Covariance Matrix Adaptation Evolution Strategy (IPOP-CMA-ES) algorithm is a reference stochastic optimizer dedicated to blackbox optimization, where no prior knowledge about the underlying problem structure is available. This paper aims at accelerating IPOP-CMA-ES thanks to high performance computing and parallelism when solving large optimization problems. We first show how BLAS and LAPACK routines can be introduced in linear algebra operations, and we then propose two strategies for deploying IPOP-CMA-ES efficiently on large-scale parallel architectures with up to thousands of CPU cores. The first parallel strategy processes the multiple searches in the same ordering as the sequential IPOP-CMA-ES, while the second one processes concurrently these multiple searches. These strategies are implemented in MPI+OpenMP and compared on 6144 cores of the supercomputer Fugaku. We manage to obtain substantial speedups (up to several thousand) and even super-linear ones, and we provide an in-depth analysis of our results to understand precisely the superior performance of our second strategy. These results are finally confirmed on a local compute cluster with 512 cores

Keywords: Parallel Optimization ; Blackbox Optimization ; Local Optimization ; Large-Scale Parallelism ; BLAS

1 Introduction

Optimization problems are prevalent in numerous modern scientific and engineering domains, necessitating increasingly intricate and compute intensive algorithms. They can also be of different nature depending on the application domain, the underlying computational complexity, the extent of information made available to the solvers, etc. This paper addresses blackbox continuous optimization problems requiring large-scale parallel architectures. More precisely, the goal is to find a real-valued solution $\mathbf{x} \in \mathbb{R}^n$ that minimizes (or maximizes) a continuous objective function $f : \mathbb{R}^n \to \mathbb{R}$. The function f is given as blackbox, meaning that no predetermined mathematical knowledge is available about the function, such as its derivatives or any information about its structure (e.g. is the function smooth or convex?). A blackbox optimization algorithm then operates by probing f for input $\mathbf{x}$, obtaining the corresponding fitness value $f(\mathbf{x})$, and proceeding accordingly for the rest of the search process in an iterative manner. Blackbox optimization problems have received considerable attention for being essential

in many application domains. These include complex engineering models or numerical simulations, and generally application fields where problem specifics remain elusive. For instance, applications in aeronautics necessitate optimizing aerodynamics through simulations of airflow around vehicles, nuclear physics involves simulating heat or particle diffusion to optimize the dimensions of containment vessels, urban transportation systems require optimizing a traffic flow determined by the patterns of stoplights, etc. Traditional gradient-based approaches cannot be applied to such problems. This led to the development of various classes of blackbox optimization algorithms, also known as derivative-free algorithms [36, 17].

We are interested here in the so-called CMA-ES (Covariance Matrix Adaptation Evolution Strategy) algorithm [27], and more specifically in its IPOP-CMA-ES[10] (Increasing Population CMA-ES) variant. CMA-ES is a state-of-the-art blackbox optimization algorithm, which is, along with its variants, the best optimizer in the GECCO black-box optimization competition [64]. CMA-ES also has applications in many fields: neural networks [42], applied physics and engineering (e.g. for the design of thermal cloaks [20], optic cloaks [21], lenses [55], gas turbines [29]), autonomous sailing [45, 44], hydrology [65], sensor networks [5], wind energy [59], solar energy [34], to cite a few. CMA-ES is an iterative algorithm. At each iteration, it samples a set of λ points in the decision space (called the population) using a probability distribution determined by the current mean point and a $n \times n$ covariance matrix, with n the dimension of the objective function f. The qualities of the λ points are evaluated by probing the blackbox function f and used to update the mean point and the matrix for the next iteration. These updates involve linear algebra operations and aim at directing the search towards interesting neighbouring areas in the decision space. The IPOP-CMA-ES[10] (Increasing Population CMA-ES) restart strategy improves CMA-ES, especially when dealing with complex problems[64]. After a CMA-ES execution (referred to as *a descent* henceforth) terminates, possibly due to being trapped in a local optimum, a subsequent CMA-ES descent is initiated with an increased population size, and this process continues iteratively. This enables IPOP-CMA-ES to employ increasingly thorough searches, at the cost of more and more function evaluations, to eventually find better optima.

CMA-ES and IPOP-CMA-ES can be used to tackle challenging blackbox optimization problems. However, large optimization problems still require a lot of compute power, because the function evaluations can be individually time-consuming, and/or because many function evaluations (i.e. CMA-ES iterations) can be needed to solve complex problems. Two main approaches can be distinguished in the literature to accelerate CMA-ES and IPOP-CMA-ES for large optimization problems. The first class of approaches focuses on reducing the cost of the linear algebra operations while attempting to maintain as much effectiveness as possible in generating new promising solutions. Instead of using a full matrix of n^2 parameters, this can be achieved by storing a reduced number of parameters, resulting in lower costs when updating and utilizing the matrix. For example, some variants can manage to store $k < n$ vectors of n parameters [38, 41, 6, 40, 39, 30], or only n parameters [60], or even solely the sampled points[33]. Subsequent works have compared the respective merits of such large-scale variants [70, 57, 69], considering their reduced ability to retain information about the local function landscape, which results in less effective convergence and lower quality of the final solution. The second class of approaches leverages parallelism, essentially performing multiple independent function evaluations [51, 11], while ensuring that the search ability is not affected. This was for example used for specific application domains [42, 18, 31]. Additionally, certain parameters can be adapted during the CMA-ES descent [11], in the hope of finding settings which best fit the considered parallel hardware. Another way of using parallelism is to run multiple CMA-ES descents concurrently, while trying to improve the convergence of any given descent using information from the other descents. In the optimization field, this method is known as the island model. Some

island models were proposed for the original CMA-ES[52, 51, 61]. Such island models have also been used in specific domains[23, 49, 54], or for multi-objective optimization[13]. Some works also implement an island model for a large-scale CMA-ES variant [7, 15, 16]. However, to the best of our knowledge, there currently exists no work on the parallelization of IPOP-CMA-ES, which implies running descents of increasing population sizes.

In this paper, we thus focus on the use of parallelism and HPC to solve large optimization problems with IPOP-CMA-ES by speeding up both the linear algebra operations and the function evaluations. Our contributions can be summarized as follows.

- We first show how the CMA-ES linear algebra operations can be accelerated thanks to BLAS and LAPACK routines. This requires the rewrite of some of these operations in order to introduce more efficient BLAS routines.
- We present two parallel strategies for IPOP-CMA-ES to fully exploit a large number of CPU cores (up to several thousands). Such a number of CPU cores implies multiple compute nodes in distributed memory, each node being composed of multiple cores in shared memory. The goal here is to leverage large-scale parallelism (via multiple nodes) to benefit from the increasing number of (parallel) evaluations in IPOP-CMA-ES. The first strategy performs descents in the same order of population size as the original IPOP-CMA-ES, while the second strategy concurrently processes descents of different population sizes.
- We thoroughly compare MPI[47]+OpenMP[58] implementations of our two strategies on 6144 cores (128 nodes) of the supercomputer Fugaku in order to determine which one is the most relevant on such a large-scale parallel architecture. We also present results obtained on top of 512 Intel Xeon cores in order to fairly support our findings when using a more conventional HPC compute cluster. Our empirical comparisons are performed using the reference BBOB[25] (Black-Box Optimization Benchmarking) benchmark configured with various dimensions and various function evaluation costs. Accordingly, we conduct a comprehensive analysis to assess the impact of the considered parallel strategies on both performance and solution quality, as a function of the target function, problem dimensionality, and evaluation costs.

The rest of this paper is organized as follows. In Section 2, we give some background on CMA-ES and on IPOP-CMA-ES. In Section 3, we show how we have introduced BLAS and LAPACK routines, and we describe the proposed parallel strategies. We report our detailed experimental study on Fugaku in Section 4, and we confirm our results on a local cluster in Section 5. We finally conclude the paper in Section 6.

2 CMA-ES with increasing population

We discuss here the main working principles of the Covariance Matrix Adaptation Evolution Strategy with Increasing Population[10] (IPOP-CMA-ES), starting with the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) it is based on.

2.1 The Covariance Matrix Adaptation Evolution Strategy

A high-level template of the CMA-ES is given in Algorithm 1. Let n be the dimension of the function f to be optimized. CMA-ES maintains a best current point $m \in \mathbb{R}^n$, also called the *mean*. At each iteration, CMA-ES samples a *population* of λ points in $\mathbb{R}^n$ using a multivariate normal distribution $\mathcal{N}(m, C)$ with a mean m and a *covariance matrix* $C \in \mathbb{R}^{n \times n}$. In other

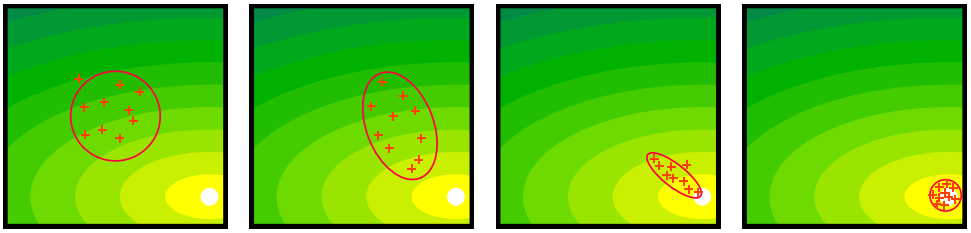


Figure 1: The CMA-ES sampling process in a decision space of dimension 2. The white dot indicates the optimum, the red ellipse the normal law, and the red crosses the sampled points.

Algorithm 1 Pseudocode of CMA-ES with population size λ

initialize (mean: m, covariance matrix: C, variance: σ, evolution path of σ: p_σ, evolution path of C: p_c)
while no stopping criterion met **do**
    ▷ *Perform one CMA-ES iteration* ◁
    for $i = 1..\lambda$ **do**
        $x_i \leftarrow \text{sample\_point}(m, C, \sigma)$
    for $i = 1..\lambda$ **do**
        $f_i \leftarrow f(x_i)$
    update $(m, C, \sigma, p_\sigma, p_c)$ using $(x_i)_{i=1..\lambda}$ and $(f_i)_{i=1..\lambda}$
return the sampled point with the best quality

words, such a distribution is an extension of the normal law centered at the mean m and distorted along the width and orientation of the multi-dimensional ellipsoid encoded by the matrix C (see Figure 1 for an illustration). This allows CMA-ES to *adapt* the search, and the underlying sampling process, to the local shape of the function it is optimizing. Specifically, the mean m, as well as the matrix C, are dynamically updated depending on the quality, that is the f-values, of the sampled points. Generally speaking, the adaptation of the mean simply consists in choosing a new mean value based on the best sampled f-values. The covariance matrix adaptation aims at updating the ellipsoid width and orientation towards the best so-far sampled points. This implies more complex algebra, having $O(\lambda \times n^2)$ complexity. In fact, in addition to the covariance matrix C, the scale at which CMA-ES searches for new points is determined by a step-size $\sigma \in \mathbb{R}$. Jointly adapting the step-size σ and the covariance matrix C is a critical ingredient for the success of the CMA-ES search process. This adaptation process is controlled using the so-called *evolution paths* $p_c \in \mathbb{R}^n$ and $p_\sigma \in \mathbb{R}^n$, respectively to C and σ. Without going into further details, the motivation behind using the evolution paths is to allow CMA-ES to learn from the points sampled so-far in the previous iterations, and to adapt the search trajectory accordingly, eventually converging more effectively towards the true optimum. To summarize, at each iteration, CMA-ES sample new points, adapts the sampling parameters accordingly, and resumes the search for the next iteration, until a stopping condition is met. The stopping condition in CMA-ES can be typically set according to search status. For further details behind all of these mechanisms, the reader is refered to the orginal design of CMA-ES[27, 10].

In the following, we will only recall the mathematical equations necessary to describe our contributions, specifically the equation for sampling new points and adapting the covariance matrix. All other algebraic considerations, such as evolution paths and step-size adaptation mechanisms, follow exactly the same specifications as in CMA-ES. These are not included here in order to keep the presentation focused. Sampling from the multi-dimensional normal

Algorithm 2 Pseudocode of IPOP-CMA-ES with initial population size λ_{start}, multiplicative factor 2, maximum coefficient K_{max}

1: $K \leftarrow 1$
2: **while** $K \leq K_{max}$ **and** *budget not exhausted* **do**
3: initialize (mean: m, covariance matrix: C, variance: σ, evolution path of σ: p_σ, evolution path of C: p_c)
4: **while** no stopping criterion met **do**
5: process one iteration of CMA-ES (see Algorithm 1, lines 4-8) using m, C, σ, p_σ, p_c and with population size $K \times \lambda_{start}$
6: $K \leftarrow K \times 2$
7: **return** the sampled point with the best quality (over all descents)

law $\mathcal{N}(0, C)$ requires two matrices B and D, where B is a matrix containing orthonormal eigenvectors of C, and D is a diagonal matrix containing the square roots of the eigenvalues of C. In CMA-ES, we eventally want to sample the point $x_k \in \mathbb{R}^n$ from the normal law $\mathcal{N}(m, \sigma^2 C)$. The corresponding equation can then be written as:

$$x_k = m + \sigma BDz_k, \quad \forall 1 \leq k \leq \lambda \tag{1}$$

where $z_k \in \mathbb{R}^n$ is sampled from $\mathcal{N}(0, I)$, with I the identity matrix. Besides, the equation for adapting the covariance matrix in CMA-ES is as follows:

$$C \leftarrow C + c_\mu \sum_{i=1}^{\lambda} w_{rk(i)}(y_i y_i^T - C) + c_1(p_c p_c^T - C) \tag{2}$$

where $c_\mu \in \mathbb{R}$ and $c_1 \in \mathbb{R}$ are the learning rate parameters for CMA-ES, $w_{rk(i)} \in \mathbb{R}$ is a weight that depends on the rank of the point x_i when sorted by it f value (points with better function values have greater weights, and $\sum_{i=1}^{\lambda} w_{rk(i)} = 1$), and $\forall i \in [\![1, \lambda]\!], y_i \in \mathbb{R}^n$ is such that $x_i = m + \sigma y_i$ [26]. Notice that computing B and D from the matrix C involves an eigendecomposition, which requires $O(n^3)$ operations. Updating other variables like p_c or p_σ requires at most $O(n^2)$ or $O(\lambda \times n)$ operations and is thus less time-consuming.

2.2 The increasing population restart strategy

CMA-ES is a stochastic search algorithm. Two executions of Algorithm 1 (also referred to as *descents*) on the same problem may thus return a different best point as output. As such, running CMA-ES multiple times can straightforwardly enhance the quality of the final best solution. The benefits behind such multiple executions depend on the shape of the objective function. Besides, to identify if the search has eventually converged and if the current execution must stop, multiple stopping conditions[9] have been proposed. The stopping conditions can generally be understood as either the function quality not improving anymore, or the function being locally too flat, or even the sampling distribution being too small (i.e. the mean stops moving, etc). In such a situation, the search can hence be *restarted* (i.e. launch a new descent) in the hope of finding better solutions.

Interestingly, it has then been shown that increasing the population size λ for each new restart enables better convergence [10] to high-quality solutions. Such an Increasing Population restart strategy has led to the design of the state-of-the-art IPOP-CMA-ES variant[10, 43, 67, 8]. In fact, IPOP-CMA-ES has been shown to provide the same convergence rate as CMA-ES on simple functions, while performling significantly better one on many complex functions. The

corresponding high-level template of IPOP-CMA-ES is presented in the template of Algorithm 2. It relies on a multiplicative factor of 2 to increase the population size at each restart. The initial population size λ_{start} is hence multiplied by $K = 2^i$ for the i-th CMA-ES execution, with K ranging from 1 to a user-defined maximum value K_{max}, e.g., $K_{max} = 2^8$. Notice that at each iteration of a given descent, multiple function evaluations are performed and the number of function evaluations equals the population size. Since these evaluations are embarrassingly parallel, IPOP-CMA-ES offers an increasing degree of parallelism along its execution. When targeting a parallel version of IPOP-CMA-ES, this increasing degree of parallelism is an opportunity to reach important parallel speedups. however, we need to design a relevant strategy to deploy at best this varying parallelism degree on a given (fixed) number of CPU cores. For this purpose, we consider in the next section different parallel strategies.

3 High performance parallel strategies

3.1 High performance linear algebra

3.1.1 Motivation

Depending on the application, the cost of linear algebra within a parallel CMA-ES algorithm may or may not be negligible. The following are examples of applications where this linear algebra cost significantly affects performance.

Firstly, there are cases where the cost of a function evaluation is lower or similar to the linear algebra one. One example is topology optimization [20], where the objective function involves a simulation using the finite element method. The cost of a single function evaluation is typically $O(N)$, and occasionally $O(N^2)$. In contrast, the cost of linear algebra operations comes down to $O(N^2)$ per iteration, since the $O(N^3)$ operations are performed once every N iterations, with N values up to 4 000.

Secondly, an entire class of functions with low evaluation cost arises in surrogate optimization. This technique is commonly used to optimize expensive objective functions. A surrogate function is constructed using machine learning. The key advantage is that the surrogate model is significantly less time-consuming to evaluate than the original objective function. For instance, in real-world problems, the computation time can be reduced: from days to milliseconds in various scientific simulations ranging from high-energy-density physics to oceanic pelagic stoichiometry modeling [35]; to 3-6 milliseconds in virtual thermal sensor simulations [22]; to milliseconds in predicting the mechanical performance of materials [71]. However, because the surrogate model has the same dimensionality as the original function, the associated linear algebra operations remain equally expensive. Moreover, while surrogate models reduce the cost per evaluation, they are not easier to optimize in terms of the number of evaluations required. For example, in real-world scenarios, optimization may require: 100,000 surrogate evaluations for vehicle dynamics control systems [68]; 24 hours for electromagnetic problems [14]; and 24 hours or more for decentralized energy systems [63]. Thus, optimizing a surrogate model can still be time-consuming and may benefit substantially from a modest or large parallel architecture. In such cases, while the majority of CMA-ES computational effort can be devoted to function evaluations, accelerating linear algebra can still yield significant performance gains. Consider indeed a scenario where CMA-ES spends in sequential 99% of its time on multiple function evaluations and only 1% on linear algebra. As per Amdahl's law, distributing the function evaluations across 99 CPU cores leads to the non-accelerated linear algebra accounting for 50% of the total runtime in parallel. In this context, accelerating linear algebra can have a strong impact on the overall CMA-ES speedup.

Therefore, accelerating linear algebra may be crucial, depending on the application. In Section 3.1.2, we thus show how we can leverage BLAS and LAPACK routines to accelerate linear algebra on a single core. Parallel computation of linear algebra will be investigated in Section 4.2.

3.1.2 Design

We now consider how standard BLAS and LAPACK routines can be incorporated into CMA-ES (and, by extension, IPOP-CMA-ES). Our primary goal here is to develop an efficient implementation of the core operations of CMA-ES, which can then serve as a reliable baseline for integrating various parallel function evaluation strategies. Let us recall that BLAS (Basic Linear Algebra Subprograms[1]) provides high-performance implementations of standard linear algebra operations: vector operations (Level 1 BLAS), matrix-vector operations (Level 2 BLAS), and matrix-matrix operations (Level 3 BLAS). LAPACK (Linear Algebra PACKage[2]) then relies on BLAS routines to efficiently solve other sophisticated operations, e.g. systems of linear equations, eigenvalue problems, singular value problems, etc. As previously presented in Section 2.1, each iteration of CMA-ES involves linear algebra operations. Our implementation is based on the serial C reference code of CMA-ES[3]. Surprisingly, this implementation does not use any BLAS or LAPACK call. In fact, matrix multiplications are written using standard loops, and the algorithm for the eigendecomposition is ultimately from Wilkinson and Reinsch [73]. In our work, we focus on Level 3 BLAS operations with a cubic time complexity for a quadratic input data size. These operations provide indeed a linear arithmetic intensity which enables their effective implementation to highly take advantage of current CPU architectures. We are then able to improve the reference C implementation of CMA-ES by introducing BLAS and LAPACK routines, either straightforwardly or thanks to some rewriting of the linear algebra operations. Specifically, this concerns three steps as discussed in more details in the following. Notice that among these three steps, we found that the first two were already implemented in the Fortran source code of the CMA-ES *pCMALib* library [51] ; but, these optimizations, and their performance impact (especially for IPOP-CMA-ES), were not presented in the corresponding paper[51]. Moreover, to our knowledge, the last step was not considered before, even when analyzing the *pCMALib* Fortran library.

- Firstly, the eigendecomposition of the covariance matrix C can easily benefit from LAPACK by using the *dsyev* routine.
- Secondly, the original equation for the covariance matrix adaptation (as given by Eq. 2) does not involve any matrix-matrix multiplication. However, since we have that $\sum_{i=1}^{\lambda} w_{rk(i)} = 1$, we can first rewrite Eq. 2 as follows:

$$C \leftarrow (1 - c_\mu - c_1)C + c_\mu(\sum_{i=1}^{\lambda} w_{rk(i)} y_i y_i^T) + c_1 p_c p_c^T .$$

  Let us then denote by M the $n \times n$ matrix equal to $\sum_{i=1}^{\lambda} w_{rk(i)} y_i y_i^T$. We then have:

$$\forall (r,c) \in [\![1,n]\!]^2, M_{r,c} = \sum_{i=1}^{\lambda} w_{rk(i)} (y_i)_r (y_i^T)_c = \sum_{i=1}^{\lambda} (y_i)_r (w_{rk(i)} (y_i^T)_c) = \sum_{i=1}^{\lambda} A_{r,i} B_{i,c}$$

[1]See: `https://www.netlib.org/blas/`
[2]See: `https://www.netlib.org/lapack/`
[3]See: `https://github.com/cma-es/c-cmaes`

where A is a $n \times \lambda$ matrix containing columns of $(y_i)_{i=1..\lambda}$ and B is a $\lambda \times n$ matrix containing rows of $(w_{rk(i)} y_i^T)_{i=1..\lambda}$, namely:

$$A = \begin{pmatrix} | & & | & & | \\ y_1 & \cdots & y_k & \cdots & y_\lambda \\ | & & | & & | \end{pmatrix}, B = \begin{pmatrix} — & w_{rk(1)} y_1^T & — \\ & \vdots & \\ — & w_{rk(k)} y_k^T & — \\ & \vdots & \\ — & w_{rk(\lambda)} y_\lambda^T & — \end{pmatrix}.$$

Therefore, we can rewritte the covariance matrix adaptation step as follows:

$$C \leftarrow (1 - c_\mu - c_1) C + c_\mu A \cdot B + c_1 p_c p_c^T \tag{3}$$

where the matrix product $A \cdot B$ can be very efficiently performed thanks to the Level 3 *dgemm* BLAS. Notice that $2\lambda n$ affectations are required to create the matrices A and B, but this cost will be dominated by the λn^2 operation cost of the matrix product: the BLAS performance gain is thus likely to offset (at least partly) these extra affectations. Moreover, the sizes of the so-defined matrices A and B depend on λ. Consequently, this makes our BLAS rewriting relevant for IPOP-CMA-ES for at least two reasons. First, the covariance matrix adaptation is more time-consuming for IPOP-CMA-ES, where the population size $\lambda = K\lambda_{start}$ is increasing with K and becomes eventually larger than for a standard CMA-ES execution. Second, since the best *dgemm* performance is obtained for large enough matrices, the BLAS gain will be stronger for IPOP-CMA-ES.

- Thirdly, regarding the sampling of new points, as previously depicted in Eq. 1, this implies only matrix-vector products (since D is a diagonal matrix in Eq. 1). However, we can generate all of the $(x_k)_{k=1..\lambda}$ sampled points at once as follows:

$$\begin{pmatrix} | & & | & & | \\ x_1 & \cdots & x_k & \cdots & x_\lambda \\ | & & | & & | \end{pmatrix} = \begin{pmatrix} | & & | & & | \\ m & \cdots & m & \cdots & m \\ | & & | & & | \end{pmatrix} + \sigma BD \cdot \begin{pmatrix} | & & | & & | \\ z_1 & \cdots & z_k & \cdots & z_\lambda \\ | & & | & & | \end{pmatrix}.$$

  In practice, this only implies λn extra affectations to fill a $n \times \lambda$ matrix with the mean vector m. Again, this extra cost will be dominated by the matrix mutliplication cost (λn^2 operations), and likely be offset by the BLAS performance gain. Moreover, as for the covariance matrix adaptation step, the matrix sizes here depend on λ which makes this step more time-consuming, and the BLAS performance gain stronger, when considering IPOP-CMA-ES.

3.2 The parallel strategies

In this section, we aim at deploying IPOP-CMA-ES on large-scale parallel architectures with thousands of CPU cores. Such architectures are based on multiple nodes (distributed-memory parallelism) composed by few multi-core processors each (shared-memory parallelism). We will thus rely on a hybrid MPI+OpenMP approach, using MPI for inter-process communications and OpenMP for multi-thread parallelism. In the reminder, we denote by T, the number of threads in each MPI process. We consider an IPOP-CMA-ES execution flow with consecutive population sizes $K \times \lambda_{start}$, K being a power of 2 ranging from 2^0 to K_{max} (see Algorithm 2). For such an IPOP-CMA-ES execution, we propose two generic parallel evaluation strategies targeting a fixed (arbitrarily large) number of CPU cores. These strategies will then be deployed in practice on a given HPC compute environment, and their relative perfomance systematically analyzed and compared in Section 4.

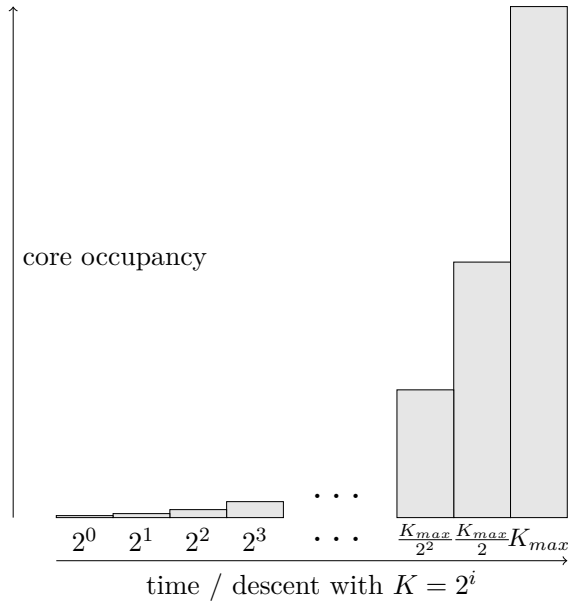


Figure 2: Illustration of the core occupancy of a naive version of IPOP-CMA-ES with successive parallel descents.

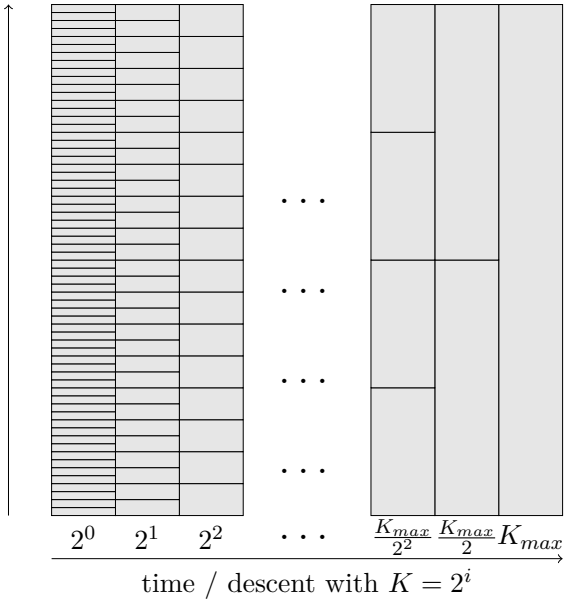


Figure 3: Illustration of the core occupancy of the K-Replicated strategy.

3.2.1 Parallelism within a CMA-ES descent

Common to all our parallel strategies, each single CMA-ES descent will be driven by a dedicated MPI process, hereafter referenced to as the *main* process. At each iteration of a given descent, CMA-ES evaluates the objective function f on λ points. These λ evaluations can be performed in parallel. In order to acheive the best possible parallel speedups, we aim at fully exploiting this parallelism level by processing each evaluation on a dedicated CPU core. Specifically, when $\lambda \leq T$, we can distribute the λ evaluations on the T threads of the main MPI process. When $\lambda > T$, we have to rely on multiple MPI processes. The main process will thus first generate the list of points where the objective function must be evaluated. These points are then "scattered" (using the corresponding MPI function) on a set of MPI processes. This sets an implicit synchronization among all processes involved in the same descent. Each of these MPI processes will evaluate the function on its points (using its T threads), and the objective function values are finally "gathered" (using again the corresponding MPI function) back to the main process. Finally, we will also consider standard multi-thread parallelism for the linear algebra operations (of Section 3.1.2) performed in each single descent. This will be detailed in Section 4.2. In the next two sections, we describe the parallel evaluation strategies.

3.2.2 The K-Replicated strategy

As illustrated in Figure 2, the first idea that may come to mind when designing a parallel IPOP-CMA-ES execution, is to successively run each descent of increasing K-value using the parallelism available within each descent (as described in the previous section). The clear drawback of such an approach is the overall low CPU core occupancy. In fact, since a descent with a large K-value have a higher degree of parallelism, most of the CPU cores will be unused for the other descents with smaller K-value.

A first solution, hereafter referred to as 'K-Replicated', is to replicate the current descent (at a given K-value) into multiple, independent descents (using the exact same K value but a

Algorithm 3 Pseudocode of the K-Replicated strategy. The global communicator MPI_COMM_WORLD (containing all MPI processes) and $Kmax$ are used for the initial call.

```
procedure K-REPLICATED(communicator, K)
    if K > 1 then
        my_rank ← MPI_COMM_RANK(communicator)
        size ← MPI_COMM_SIZE(communicator)
        my_half_comm ← MPI_COMM_SPLIT(communicator, my_rank ≤ size /2,
        my_rank)                    ▷ splits 'communicator' in two halves of equal size
        K-REPLICATED(my_half_comm, K/2)
    if K ≤ K_max then
        CMA-ES DESCENT(communicator, K × λ_start)
```

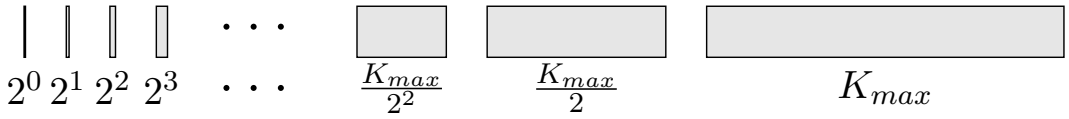


Figure 4: Illustration of the K-Distributed algorithm.

different seed) until all the computing resources (cores) are used This is illustrated in Figure 3. The number of replicated descents is then $c/(K \times \lambda_{start})$, where c is the total number of CPU cores available. This strategy implies more simultaneous parallel descents at the begining, when K is small, and fewer descents when K is larger. On one hand, a full usage of the CPU cores is guaranteed. On the other hand, since CMA-ES is a stochastic algorithm, the additional replicated descents can help finding better solutions. In other words, the K-replicated strategy exploits all the available CPU cores by replicating multiple descents with the same K-value. These concurrent descents increase the chances of finding better solutions for the stochastic IPOP-CMA-ES algorithm.

Regarding the implementation, we notice that once two K descents are finished, their assigned resources can be used for a subsequent $2K$ descent. This can be efficiently implemented as in the recursive Algorithm 3 using a hierarchy of MPI communicators, which represent sets of MPI processes that can exchange messages [47]. New communicators can be created by specifying subsets out of a previously existing communicator; Each process of the parent communicator indicate which child communicator it belongs to. The global communicator is thus split until the resulting communicators are small enough to be used for descents with $K = 1$. Then, each time a pair of $K = 2^i$ descents from a same parent communicator is finished, the control flows back up to this parent communicator for a $K = 2^{i+1}$ descent. This is repeated until K_{max} is reached. In order to have a distinct random generator seed in each descent, we rely on the current time multiplied by the rank of the MPI process in the global communicator.

3.2.3 The K-Distributed strategy

We now consider a second strategy to leverage parallelism among multiple CMA-ES descents for large-scale parallel architectures, which stems from three observations. Firstly, a descent using a population size λ can be sped-up up to a factor of λ (provided the communication and linear algebra times are short enough). Secondly, one may assume that a descent using a population size $K \times \lambda_{start}$ may take, roughly, K times as long to reach a given quality as when using a poulation of size λ_{start}. This can be interpreted as CMA-ES taking in more information about a local landscape of the function before making the choice of where to move next, which requires

more time. The benefit of using a larger population size is that, in many cases, the decision will be more 'informed', causing CMA-ES to avoid being trapped in a local optimum for longer, and ultimately finding better solutions before the end of the descent. Finally, although the previous assumption seems reasonable, from a very general perspective, there could also be cases where, for some quality ranges on some objective functions and for some K-values, a $K \times \lambda_{start}$ descent will be faster or slower than K times the duration of a λ_{start} descent. This is because the optimal population size should depend on the properties of the function's landscape. Note that, for complex functions, the local landscape properties can change depending on the region of the search space or on the scale it is observed at.

From the first and second observation, we can reasonably expect that several descents, each with a different population size λ and each running on λ CPU cores, will generally improve the quality of their current best solution for roughly the same amount of computation time. Then, from the third observation, we can expect that, thanks to their different population sizes, some of these parallel descents will be faster than others to reach certain qualities. Therefore, running a range of population sizes in parallel may provide better results than running several descents with the same population size, as K-Replicated does. This leads us to consider running concurrently all descents with a distinct K value each. As illustrated in Figure 4, the $\log_2(K_{max}) + 1$ descents are executed at the same time, each with a population size equal to $K \times \lambda_{start}$, for $K = 2^0, 2^1, 2^2, .., K_{max}$. We refer to this algorithm as 'K-Distributed'. We implement it with MPI by splitting the initial communicator into $\log_2 K_{max} + 1$ sub-communicators, each containing twice as many processes as the previous one.

4 Experimental results on Fugaku

In this section, we conduct a step-by-step performance analysis on the supercomputer Fugaku of the different parallel and high performance strategies described before. We first start describing our experimental set-up including the setting of the optimization benchmark functions, the considered parallel computing environment and how our parallel strategies are specified for the Fugaku hardware. Then, we report the benefits of using sequential and multi-threaded BLAS/LAPACK routines for CMA-ES. Our main results dealing with the different parallel strategies are then reported and analyzed in a comprehensive manner.

4.1 Experimental setup

Firstly, for the purpose of comparing the considered algorithms from a pure optimization perspective, we consider the COmparing Continuous Optimizers (COCO) [28] framework providing an implementation of the Black-Box Optimization Benchmarking (BBOB)[4] test suite [25]. BBOB is a state-of-the-art blackbox test suite, used in particular in a reference workshop held yearly in the well-established Genetic and Evolutionary Computation Conference (GECCO). More than 200 algorithms were already benchmarked within this framework. BBOB provides a set of 24 continuous functions exposing different properties believed to represent a relatively broad range of blackbox optimization problems that one may encounter in practice. All functions are available in multiple dimensions: in this paper we will study dimensions 10, 40, 200 and 1000. It is to be noticed that algorithms benchmarked using the BBOB functions, and the underlying COCO framework, usually fix as a budget the total amount of function evaluations that an algorithm is allowed to query. However, although the time it takes to query one blackbox function evaluation may vary across different BBOB functions of different dimensions, the BBOB test suite does not allow to explicitly control the evaluation times, which are in fact

[4]See also: `https://numbbo.github.io/data-archive/bbob/`

very short (less than 9ms in dimension 1000 on average across all functions). In practice, the time to evaluate a blackbox function directly impacts the overall CPU time required to run an algorithm. It is hence an important feature to account for when fairly assessing the performance of a parallel optimization algorithm. In fact, this has a direct impact on the computation grain size (i.e. the amount of work performed by each "task" in parallel). Therefore, in our work, and in addition to the broad range of functions provided by the BBOB test suite, we also consider to accommodate different blackbox evaluation times by adding artificial additional times to the BBOB function evaluations. For dimensions 10 and 40, for which BBOB functions have low evaluations costs, we consider additional costs of 1ms, 10ms and 100ms. This will allow us for a more comprehensive and realistic performance assessment of the considered parallel algorithms. In fact, having such evaluation times, even for small dimensions, is commonly encountered in function optimization. For instance, Roussel et al.[37], use CMA-ES for parameter estimation with objective functions of dimension 8 and evaluation costs in the order of magnitude of the second. Evaluation costs may be even larger when scientific simulations or neural networks are involved. For example, using CMA-ES to find hyper-parameters for neural network training can necessitate 5 or 30 minutes in dimension 19[42]. Other examples of evaluation times include: groundwater bioremediation (about 8 minutes)[50], aerodynamics of a shape (about 3 or 11 minutes)[32], molecular docking (4 hours)[46], automotive crash simulation (about 17 or 29 hours)[19] ; and neural network trainings for computer vision (0 to 30 minutes)[12] or for document classification (average of 2.5 or 5.8 hours)[66].

Secondly, for the purpose of studying the parallel performance of our algorithms on large-scale parallel architectures with thousands of CPU cores, we consider running our algorithms on top of the supercomputer Fugaku. In June 2024, Fugaku was the fourth most powerful supercomputer in the TOP500 list[3], and the first in the world in the HPCG list[2] and in the GRAPH500 BFS list[1]. This massively parallel supercomputer contains 158,976 A64FX CPUs, which are ARM-based architectures developed by Fujitsu with 48 compute cores and 4 assistant cores each[56, 62]. The A64FX is divided in 4 CMGs (core memory groups) of 12 cores each. Each CMG is a NUMA (non-uniform memory access) node. Indeed, the memory space of the CPU consists of a set of HBM2 high-bandwidth memory and 2 levels of cache for each of the 4 CMGs. The CMGs communicate between each others and with the network using a ring bus. The A64FX CPUs are connected by the Tofu Interconnect D [4], a 6D torus topology. For our experiments, we consider using 128 A64FX CPUs of the Fugaku, representing a total of 512 CMGs and 6,144 compute cores.

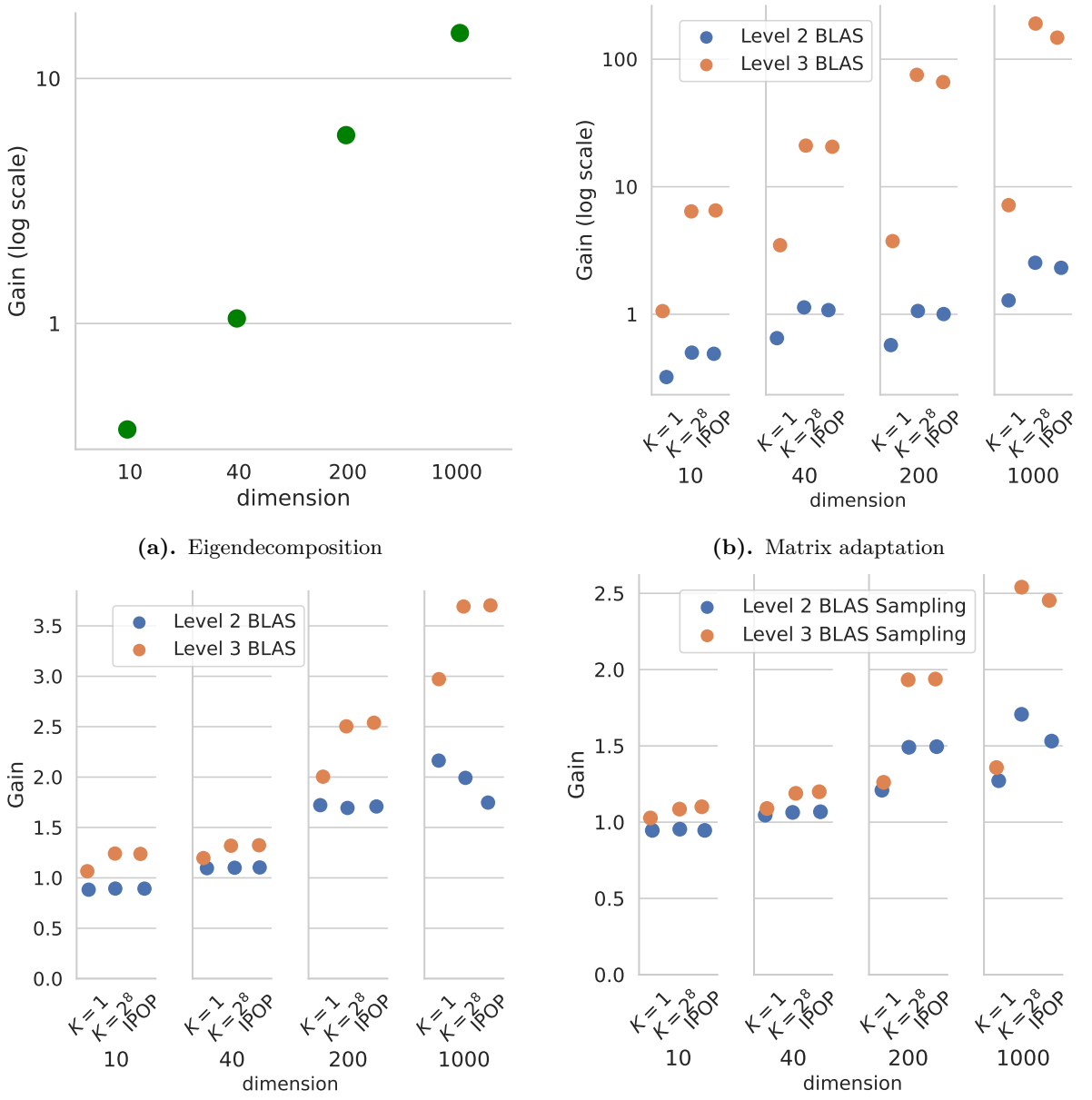
Regarding our implementations, they are all based on the serial C reference code of CMA-ES[5], which we modified to integrate the BLAS/LAPACK routines and the MPI+OpenMP parallelization. They are compiled with the Fujitsu C Compiler (version 4.11.1), the Fujitsu OpenMP and MPI libraries, and the Fujitsu thread-parallel implementation of BLAS/LAPACK. We let the C reference code set default values for all parameters, except for the initial mean m, the initial variance σ and λ_{start}. In order to better adapt to the BBOB search space, we indeed set at the start of each CMA-ES descent the initial mean m to a point selected uniformly at random in the BBOB search space, and the initial variance σ to 1/4 of the search space width. Regarding λ_{start}, the usual setting is of the order of magnitude of ten. In order to obtain the best parallel speedups and to compare our strategies on a large number of CPU cores within a single setting, we target the processing of each evaluation on a dedicated CPU core (see Section 3.2.1). For our MPI+OpenMP performance tests on Fugaku, we thus choose to have $\lambda_{start} = 12$. This way we can have $T = 12$ threads in each MPI process: the $K \times \lambda_{start}$ evaluations of a K descent are thus performed with K MPI processes with T threads each. Each A64FX runs 4 such MPI processes, i.e. one per CMG as usually done with NUMA architectures. For K-

[5]See: `https://github.com/cma-es/c-cmaes`

Replicated we set K_{max} to 2^9 which leads to $K_{max} \times \lambda_{start} = 2^9 \times 12 = 6144$ parallel evaluations executed on 6,144 cores (512 CMGs, 128 A64FX) for the final descent. For K-Distributed, we set K_{max} to 2^8 which leads to $(\sum_{i=0}^{8} 2^i) \times \lambda_{start} = 511 \times 12 = 6132$ parallel evaluations on 511 CMGs. The K-Distributed strategy thus uses 12 fewer cores than the K-Distributed one, but this is the fairest comparison we can make between these two strategies. We let the sequential IPOP-CMA-ES execute with $K_{max} = 2^9$, and we ensure that this is the sole process running on the CMG of its core, so as to prevent cache interference with other processes. To keep our experiments manageable in a reasonable time, the execution limit is set to 12 hours of wall-clock time. Except for Section 4.2 (BLAS/LAPACK tuning), we conducted 20 runs for each function and each strategy in dimensions 10 and 40. Due to time constraints, we performed at least 5 runs for each in dimensions 200 and 1000.

4.2 Linear algebra performance results

We start our analysis by studying the performance impact of introducing the BLAS/LAPACK routines in three linear algebra steps described in Section 3.1.2. Figure 5 shows the performance gains with respect to each step, each step being executed sequentially for various dimensions and for various K-values with $\lambda_{start} = 12$. Figure 5a reports the performance gain of using LAPACK specifically with respect to the eigendecomposition operation in CMA-ES. LAPACK enables us to accelerate the eigendecomposition step for problems of dimensions 40 and above, and significantly for problems of dimensions 200 et 1000 (up to 15.3x), where the C matrix is large enough to benefit from the LAPACK performance optimisations. Notice that for a relatively small dimension of 10, we found that using LAPACK leads to a performance loss, which is because the matrices maintained by CMA-ES are so small for such a dimension. However, since in dimension 10 the eigendecomposition accounts for only 9% of the overall linear algebra runtime (averaged over the 24 functions of BBOB), the overall loss in performance is negligible. Figure 5b presents BLAS performance gains with respect to the adaptation of the C covariance matrix. We distinguish here gains obtained when directly using Level 2 BLAS in Eq. 2 (see Section 2.1), and gains obtained with Level 3 BLAS thanks to our rewriting proposed in Section 3.1.2. While the use of Level 2 BLAS does not offer performance gains in dimensions 10, 40 and 200, our new computation scheme based on Level 3 BLAS offers very significant performance gains (up to 190x), especially for higher problem dimensions. The extra affectations (see Section 3.1.2) are thus offset by the BLAS gain, even for the lowest dimension. As for the sampling operations, as reported in Figure 5c, we observe that using Level 2 routines directly in equation 1 (see Section 2.1) can only provide some gain for dimensions greater than 10. However, when the operations are rewritten using Level 3 routines (see Section 3.1.2), we are able to accelerate the reference C code for any dimension, with gains stronger than for Level 2 BLAS. Again the extra affectations (see Section 3.1.2) are offset by the BLAS gains. Finally, in Figure 5d, we report performance gains due to the sampling operations, but this time in a different context. The gain is computed with respect to *all* the linear algebra part (i.e. both sampling at lines 4-5 and update at line 8 in Algorithm 1), and not just to the sampling step like in Figure 5c. The eigendecomposition uses LAPACK and the matrix adaptation uses Level 3 BLAS, whereas Level 2 or Level 3 BLAS are used for the sampling. Although the gains obtained solely for the sampling may be deemed relatively small compared to the ones obtained solely for the covariance matrix adaptation, using Level 3 BLAS for the sampling operations still has a relatively substantial impact when LAPACK and BLAS routines are already used to optimize the other linear algebra steps. For instance, this enables us to increase the overall gain over the C reference code from 1.5 to 2.5 for all the linear algebra operations in dimension 1000. As a final remark, regarding the sampling and the adaptation steps, one can see stronger



(a). Eigendecomposition

(b). Matrix adaptation

(c). Sampling

(d). Overall linear algebra part

Figure 5: a) Performance gains for the eigendecomposition of the C matrix when using LAPACK over the reference C code (written without LAPACK). b) , resp. c) Performance gains for the adaptation of the C matrix (resp. for the sampling) when using Level 2 or Level 3 BLAS over the reference C code (without BLAS). d) Performance gains over the reference C code (without BLAS and LAPACK) for all the linear algebra part, with LAPACK for the eigendecomposition and Level 3 BLAS for the C matrix adaptation, when using Level 2 or Level 3 BLAS routines for the sampling. The IPOP columns correspond to a IPOP-CMA-ES execution with successive descents using K from 1 to 2^8.

gains for $K = 2^8$ and for IPOP-CMA-ES than for $K = 1$. This is due to the larger population sizes, which lead to larger matrices. This shows that BLAS/LAPACK routines are even more relevant for IPOP-CMA-ES than for CMA-ES, the IPOP-CMA-ES increasing population sizes becoming eventually larger than the CMA-ES ones.

For completness, we report in Table 1 the proportion of CPU time linear algebra operations consume relative to the total execution time of IPOP-CMA-ES, with and without Level 3

Table 1: Proportions (averaged over all BBOB functions) of the linear algebra runtime within the overall runtime of a sequential execution of IPOP-CMA-ES (with $\lambda_{start} = 12$ and $K_{max} = 2^8$) as a function of whether it is accelerated using Level 3 BLAS and LAPACK.

	Dimension			
Accelerated	10	40	200	1000
no	38%	36%	44%	69%
yes	33%	21%	18%	21%

BLAS / LAPACK routines. As we can see, BLAS/LAPACK becomes increasingly effective at reducing the proportion of linear algebra computations as the dimension increases. High dimensionnality is a key factor that makes an optimization problem hard, making our linear algebra rewrites particularly valuable for tackling harder problems. Thanks to our BLAS/LAPACK rewrites, the linear algebra part is now minority in the overall IPOP-CMA-ES runtime. Using additional costs for the evaluations (see Section 4.1) will make the linear algebra part even more minority.

Finally, we tuned our BLAS/LAPACK implementations to determine the optimal number of threads (up to a maximum of 12) for each dimension. We found that dimensions 10 and 40 run best with 1 thread, dimension 200 with 4 threads and dimension 1000 with 12 threads. This aligns with the observation that larger dimensions entail larger matrices, which can be effectively managed by BLAS/LAPACK with a greater number of threads. However, the sizes of the involved matrices are not large enough, and the best speedup we obtain for running BLAS/LAPACK on multiple threads is 1.4×, for dimension 1000 with 12 threads. Due to this limited speedup , we believe that we would not benefit from distributing the linear algebra operations over multiple MPI processes (i.e. over more than 12 cores). That is why we chose to perform the linear algebra operations in parallel using only multi-threading within one MPI process (the main one for each descent, see Section 3.2.1), and up to 12 threads.

Now that we have accelerated the linear algebra operations, with BLAS/LAPACK routines and as much as possible via parallelism, the time spent on function evaluations is the majority in the IPOP-CMA-ES execution time; hence, making the relevant parallelization of function evaluations of high importance. We will thus now focus on our two proposed parallel strategies.

4.3 Parallel performance results

In this section, we delve into the performance behavior of the proposed K-Replicated and K-Distributed parallel algorithms. A thorough and fair performance assessment of our parallel evaluation variants requires to discuss two aspects. Firstly, although the function evaluation time can significantly influences the overall performance, it cannot be explicitly controlled in the COCO implementation of the BBOB functions. Hence, one has to adopt a more robust approach to assessing parallel performance relative to function evaluation time. Secondly, due to the stochastic nature of the considered algorithms, K-Replicated and K-Distributed are not expected to deliver exactly the same output as the sequential IPOP-CMA-ES. Consequently, it is essential to carefully define a metric that allows us to fairly evaluate the ability of the different algorithms to reach high-quality solutions within reduced time-frames. In the next subsection, we begin by discussing the methodology we adopt to address these two aspects. Subsequently, we present our findings and state our main results.

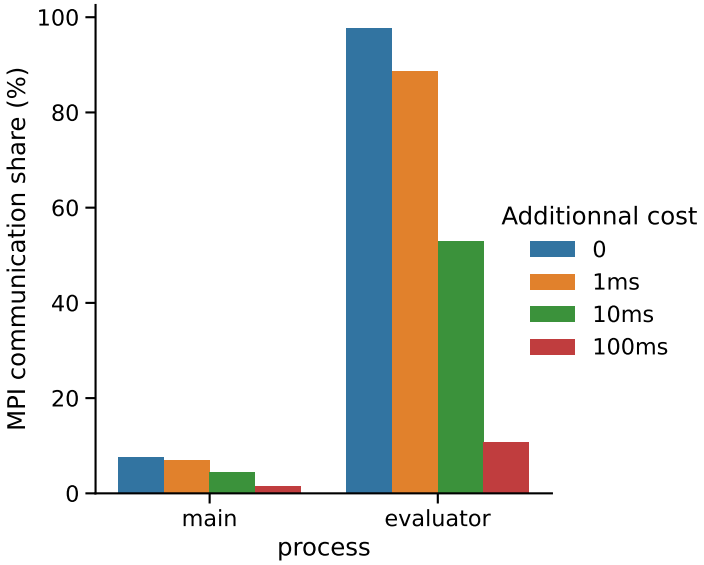


Figure 6: MPI communication shares with respect to the total runtime, as measured by the Fugaku Instant Performance Profiler (FIPP) [72] for a $K = 2^8$ descent with 256 MPI processes, averaged over all BBOB functions of dimension 40. 'main' is the main MPI process (namely, the one with rank 0) driving the $K = 2^8$ descent (see Section 3.2.1) and processing the linear algebra operations (see end of Section 4.2), whereas 'evaluator' is one of the MPI processes performing only evaluations (here, the one with the highest MPI rank). Contrary to Table 1, we consider only $K = 2^8$ and all the evaluations are performed in parallel.

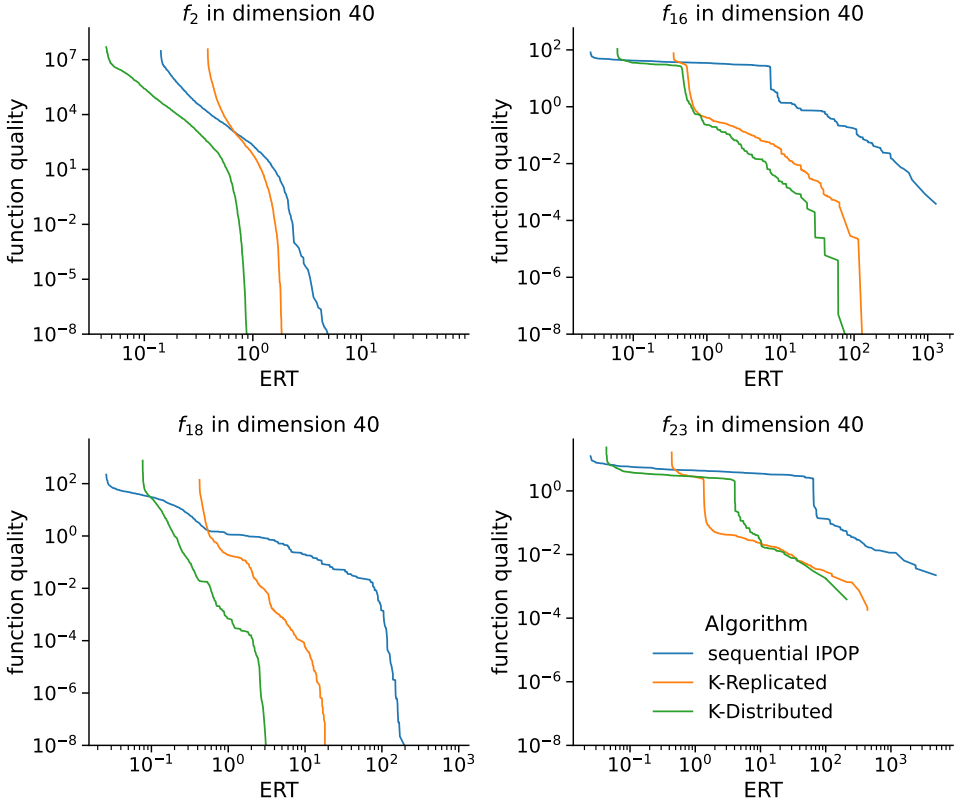


Figure 7: Function quality of the best solution found over runtime, averaged over 20 runs using the Expected Runtime (ERT) [24].

4.3.1 Performance assessment methodology

Function evaluation time. We first analyze the relevance of the additional costs introduced in Section 4.1, by reporting in Figure 6 the share of MPI communications in the total runtime of a $K = 2^8$ descent involving 256 MPI processes. When considering zero additional cost, the time spent in MPI communications (scatter and gather operations) is limited for the main process, but is the vast majority of the total time for an other process involved in the parallel descent. This is because the linear algebra part is only performed in the main process and leads to important waiting times for processes other than the main one in their MPI communications (namely at the scatter level). The linear algebra part can thus be a potential performance bottleneck when scaling on a large number of CPU cores. When adding extra costs in the function evaluation, one can see in Figure 6 that the MPI communication shares (i.e. the relative cost of the linear algebra part with respect to the total time, as well as the data transfers themselves) strongly decrease, until becoming a minority. Hence, these extra costs enable us to also 'simulate' real-life cases (see Section 4.1) where the linear algebra should not be a performance bottleneck.

Solution quality. Since the optimal solutions of the BBOB functions are known, we can evaluate the quality of a solution relative to the optimal one. In our work, we measure quality by the difference ϵ between the function value of the best solution found so far by an algorithm and the function's optimal value. Since the considered algorithms are stochastic, we get inspiration from the so-called Expected Runtime (ERT) [24] in order to aggregate the quality results obtained from multiple runs using different seeds for a same algorithm. The ERT defines the empirical average time (over the different seeds) it takes for an algorithm to hit a solution of a given target quality ϵ. Notice that two scenarios can happen. In the case all the runs of an algorithm were successful to hit the target quality ϵ, the ERT is simply the average of the hitting times. In the case some runs were unsuccessful within the maximum affordable budget, we can assume that the algorithm could have been restarted for a new run until a success is observed. The time until a new run is successful can then be viewed as a random variable, which average value can be empirically estimated in a straightforward manner using the data available from the (other) successful runs at hand. Hence, the ERT value is simply defined as the sum of the execution times of all runs (including unsuccessful ones) divided by the number of successful runs. Notice that for the ERT to be correctly defined, at least one run must be successful. The reader is referred to [24] for more details.

Furthermore, it is well known that the relative behavior of an optimization algorithm depends on the tackled function, i.e., no algorithm is better than all others for all functions. Besides, *not* all algorithms can hit the same range of targets on *all* functions. Some algorithms are even not able to reach the same target that other algorithms can reach for the same function. Consequently, this raises a number of qualitative and quantitative questions, i.e., which algorithm is able to reach a given quality? which algorithm reaches it faster than the others ? what relative speedup can be obtained for a given solution quality? etc. In particular, parallel speedups cannot be defined in the conventional manner used in parallel computing. Instead, we consider nine fixed target quality values, namely $\epsilon \in \{10^2, 10^{1.5}, 10^1, 10^{0.5}, 10^0, 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}\}$. These values are the same as the ones used in the COCO framework [28]. They are intended to represent a range of target quality going from easy, to moderate and difficult to achieve. For each given pair of BBOB function and target quality ϵ, we can then fairly analyze the relative ERTs achieved by the considered algorithms. In the following section, we define the speedup achieved by an algorithm over another one, with respect to a given BBOB function and a given target quality ϵ, as the ratio of the ERTs achieved by the two algorithms.

Table 2: Speedups obtained by the two parallel strategies over the sequential IPOP-CMA-ES, aggregated over the different pairs of BBOB functions and target qualities, for various function dimensions and function granularities. '<' (resp. '>') stands for the number of function-target couples for which K-Replicated reaches the target quality before (resp. after) K-Distributed. Function-target couples are accounted for when both parallel algorithms reach the target quality, so the sum of the counts may vary with the dimension and the granularity.

Dimension	10	10	10	10	40	40	40	40	200	1000
Additional cost	0	1ms	10ms	100ms	0	1ms	10ms	100ms	0	0
K-Replicated										
avg. speedup	1.1	83	159	219	8.6	70	160	176	59	23
std. dev.	4.0	234	432	639	24	171	520	618	183	55
min. speedup	0.1	0.1	0.6	3.7	0.0	0.1	0.5	3.9	0.1	0.1
max. speedup	30	1620	2995	5018	206	1182	3861	5121	1614	425
</>	13/182	15/182	10/188	23/170	9/173	9/174	13/170	35/148	28/134	20/107
K-Distributed										
avg. speedup	2.7	115	201	169	17	171	419	736	392	70
std. dev.	3.2	259	378	255	39	302	799	2884	1580	257
min. speedup	0.5	0.6	2.1	7.7	0.3	0.7	1.8	8.0	0.3	0.5
max. speedup	20	1610	1901	2071	267	1645	3857	18080	13944	2397

4.3.2 Overall parallel speedup

In Table 2, we summarize some basic statistics concerning the speedups obtained by our two parallel strategies over the sequential IPOP-CMA-ES. This sequential IPOP-CMA-ES leverages our Level 3 BLAS / LAPACK rewrites (with one thread) so as to focus here on the speedups obtained thanks to MPI+thread parallelism. More precisely, Table 2 reports the average, the standard deviation, the minimum and the maximum, speedups obtained respectively by K-Replicated and K-Distributed over the different pairs of BBOB functions and target qualities. The results are reported for each considered function dimension and function granularity. Additionally, the row "</>" of Table 2 offers a direct comparison : each function-target pair is counted in the left-hand (resp. right-hand) number when K-Replicated reaches the target quality before (resp. after) K-Distributed. For example, the first cell with value 13/182 reads as K-replicated has better ERT than K-Distributed on 13 function-target pairs, whereas K-Distributed has better ERT than K-replicated on 182 function-target pairs. Two main observations can be extracted from Table 2.

Firstly, we can clearly see that K-Distributed (despite using a little less CPU cores) provides almost always a better average speedup than K-Replicated. The only exception is for dimension 10 with 100ms additional cost. Interestingly, even in such a case, as shown in the "</>" row, the number of function-target pairs where K-Distributed is faster than K-Replicated is substantial. This holds true with no exception independently of the setting of the dimension and the additional cost. Moreover, we managed to obtain very high maximum speedups (over functions and targets) for both strategies, as soon as the computation grain or the dimension are large enough. It is also interesting to note that we even obtain 'super-linear' speedups for K-Distributed. This happens for dimension 40 with a function additional cost of 100ms, as well as for dimension 200, where the maximum observed speedup of K-Distributed is respectively 18080× for function f_7 with target 10^{-6}, and 13944× for f_{18} with 10^{-2}, which is substantially larger than the 6144 cores used. One should indeed recall that the considered parallel strategies do *not* imply exactly the same search behavior as serial IPOP-CMA-ES. Hence, these results support the fact that, for some specific functions/targets, the considered parallel strategies are able to show better search behavior than the original serial algorithm. Secondly, we can clearly see in Table 2 that the function evaluation granularity as captured by the considered additional

Table 3: ECD value reached by each algorithm for the final timestamp of K-Distributed for various dimensions and granularities.

Dimension	10	10	10	10	40	40	40	40	200	1000
Additional cost	0	1ms	10ms	100ms	0	1ms	10ms	100ms	0	0
Sequential IPOP	72%	31%	24%	21%	67%	34%	34%	33%	48%	39%
K-Replicated	29%	82%	83%	83%	75%	74%	78%	78%	65%	57%
K-Distributed	82%	82%	83%	82%	78%	79%	79%	80%	75%	64%

costs has a deep impact on the obtained speedups. Except for one case (when increasing the additional cost from 10ms to 100ms in dimension 10 for K-Distributed), the speedup of the parallel strategies over the serial algorithm increases indeed consistently with the function evaluation cost.

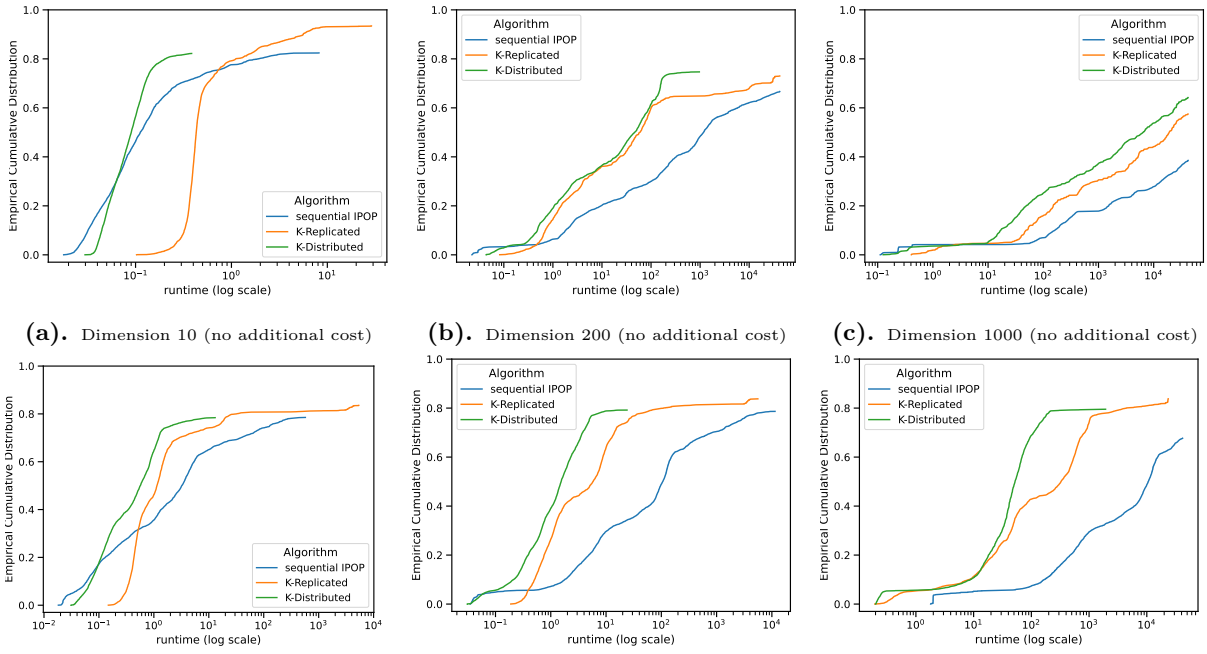
At this stage of the presentation, let us remark that although Table 2 provides a global view of the achieved speedups, the reported statistics are still to be very carefully interpreted. In particular, computing a speedup value can only be performed when *both* the sequential IPOP-CMA-ES and the parallel strategy were successful in hitting a given target. This means that the average speedup values shown in Table 2 discard the pairs of function/target where at least one algorithm was not able to hit the considered target. Generally speaking, we observed that the harder a target is, the more likely it is for serial IPOP-CMA-ES to be unsuccessful. This is exactly why the overall average speedup values reported in Table 2 decreases when going from dimension 200 to dimension 1000. However, such a decrease is *not* to be attributed to a parallel performance loss as the problem dimension increases. It is instead to be attributed to the fact that many target values cannot be hit by serial IPOP-CMA-ES. Hence, a more fine grained assessment of the behavior of the different parallel strategies is needed to fully appreciate the benefits of the designed strategies, in particular as a function of problem dimension. This is to be studied in more detail in the next section.

4.3.3 Empirical cumulative distribution analysis

In this section, we analyze the relative performance of the considered algorithms using the so-called Empirical Cumulative Distribution Functions (ECDF) [48] as introduced in the COCO benchmarking framework[6]. An empirical (cumulative) distribution function $F : \mathbb{R} \to [0, 1]$ is defined for a given real-valued data set, such that $F(t)$ equals the fraction of elements in the data which are smaller than or equal to t. In an optimization setting, the ECDF is to be viewed as a measure of how many 'problems' a stochastic optimization algorithm can solve on average for a given time budget t. For the purpose of our analysis, we consider the data-set containing the (function,target,run)-triplets labeled with the timestamp at which the algorithm was able to find a solution hitting a specified target value. The ECDF then counts for every timestamp t the proportion of (function,target,run)-triplets labeled with a timestamp smaller than or equal to t. In other words, the ECDF counts the proportion of targets that an optimization algorithm is able to hit as a function of the elapsed time t, i.e., the higher the proportion, the more powerful an algorithm.

The ECDF curves across all dimensions and without an additional cost are reported in Figures 8, a) b) c) d). Notice that an ECDF curve positioned to the left of another one indicates a more powerful algorithm. Notably, the K-Distributed's curve is almost always the leftmost which suggests that, without specific knowledge of a function landscape, K-Distributed is the superior choice. Similarly, the K-Replicated's curve is to the left of the sequential IPOP-CMA-

[6] `https://numbbo.github.io/coco-doc/perf-assessment/\#empirical-distribution-functions`



(a). Dimension 10 (no additional cost) **(b).** Dimension 200 (no additional cost) **(c).** Dimension 1000 (no additional cost)

(d). Dimension 40 (no additional cost) **(e).** Dimension 40, additional cost of 1ms **(f).** Dimension 40, additional cost of 100ms

Figure 8: ECDF (i.e. rates of function-target couples reached for a given runtime) of each algorithm for various dimensions and granularities.

ES one for most of the execution time. This ECDF analysis thus confirms the conclusions from Section 4.3.2: both parallel strategies outperform the sequential IPOP-CMA-ES, with K-Distributed being the most effective.

Next, we analyze the dynamics of the algorithms with respect to function dimension. Firstly, higher dimensions show a larger gap between the parallel variants and the sequential IPOP-CMA-ES, leading to greater speedups for the parallel variants. Secondly, for each dimension, there is a cross-over ECD value where each parallel curve crosses and stays to the left of the sequential curve, meaning the parallel variant is generally better than sequential IPOP-CMA-ES for solving problems past this point. Sequential IPOP-CMA-ES is therefore faster only for the very easiest targets and, as the dimension increases, these cross-over values decrease, making the parallel variants the better choice for a broader range of problems. Thirdly, in Table 3, we report the ECD values for each algorithm at the final timestamp of the K-Distributed strategy. We observe that ECD values decrease with increasing dimension, especially at dimensions 200 and 1000. However, for these high dimensions, the parallel strategies show greater ECD values than the sequential IPOP-CMA-ES, K-Distributed having the greatest ones. Thus, as dimension increases (which makes optimization problems more challenging), the benefits of our parallel variants, particularly K-Distributed, become more pronounced. Furthermore, in Figures 8 d) e) f), we report ECDF curves for different granularities at dimension 40 (dimension 10 leads to similar results). As with dimension, a higher granularity widens the gap between the parallel strategies and the sequential IPOP-CMA-ES. A higher granularity increases the gap between K-Distributed and K-Replicated for ECD values greater than 0.4, making K-Distributed a better choice for more time-consuming problems.

Finally, we observe specific effects in certain dimensions and granularities. In dimension 1000 (see Figure 8c), the sequential IPOP-CMA-ES stops at a lower ECD value due to the time limit, which prevents us from computing parallel speedups for many targets hit by our parallel strategies (see Section 4.3.2 and the lower average speedups for dimension 1000 in Table 2).

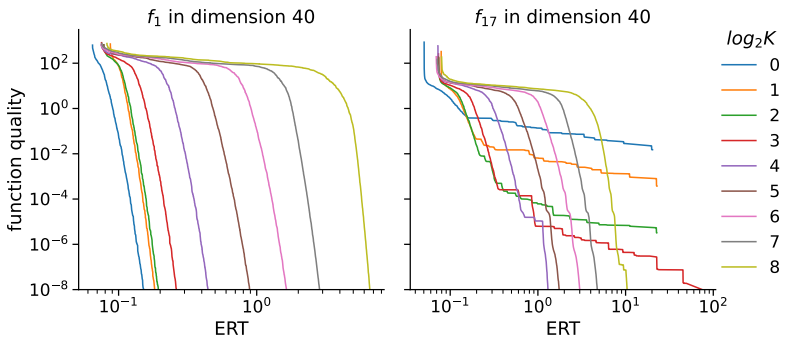


Figure 9: Quality impact of different K-values for K-Distributed.

Table 4: log_2K (averaged over 20 executions) of the first descent to reach a given quality for a K-Distributed run in dimension 40 (no additional cost). '-' indicates that no descent reached the target.

	targets								
function	10^{2}	$10^{1.5}$	10^{1}	$10^{0.5}$	10^{0}	10^{-2}	10^{-4}	10^{-6}	10^{-8}
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
2	2.2	2.4	2.4	2.7	2.8	3.0	2.8	2.8	2.7
3	0.7	3.0	-	-	-	-	-	-	-
4	1.3	5.1	-	-	-	-	-	-	-
5	0.1	0.1	0.1	0.0	0.1	0.1	0.1	0.1	0.1
6	0.1	0.1	0.1	0.3	0.4	1.1	1.0	1.2	1.3
7	0.3	0.6	1.9	3.9	4.5	4.8	4.8	4.8	4.8
8	0.3	0.7	1.4	1.6	1.6	1.7	1.8	1.8	1.9
9	0.3	0.8	1.7	1.9	1.9	2.0	2.0	1.9	1.9
10	1.8	1.6	1.8	1.7	1.9	2.0	2.0	2.0	2.1
11	3.0	2.9	2.9	3.0	2.8	2.6	2.6	2.5	2.6
12	0.2	0.3	0.7	0.9	1.1	1.2	1.2	1.6	1.6
13	0.0	0.1	0.1	0.7	1.2	2.4	2.6	3.1	3.1
14	0.1	0.0	0.0	0.0	0.0	0.1	1.5	2.4	2.6
15	0.6	2.1	4.2	5.6	6.4	6.5	6.5	6.5	6.5
16	0.5	1.0	1.2	0.9	1.6	6.9	7.2	7.5	7.0
17	0.0	0.0	0.0	0.0	0.3	1.5	2.4	3.5	4.0
18	0.1	0.1	0.1	0.5	1.2	3.7	5.4	5.7	5.7
19	0.0	0.0	0.1	1.6	1.9	-	-	-	-
20	0.0	0.1	0.1	0.0	6.9	-	-	-	-
21	0.6	0.1	0.1	2.0	2.5	2.3	2.3	2.3	2.3
22	0.0	0.1	1.4	1.4	1.0	-	-	-	-
23	0.2	0.2	0.2	5.0	1.9	3.3	-	-	-
24	1.0	4.2	5.0	5.7	-	-	-	-	-

However, past the last ECD value reached by sequential IPOP-CMA-ES, the slope of the curves for the two parallel strategies do not bend, showing that these strategies are actually highly efficient in high dimension. This is why, along with the parallel speedups, ECDF profiles are necessary to appreciate the full extent of the performance of our strategies. Notice that, in dimensions 10 and 40 for all granularities, K-Replicated's final runtime reaches slightly more targets than K-Distributed, but only long after K-Distributed is over. This is because we have let K-Replicated run up to $K_{max} = 2^9$ (as opposed to $K_{max} = 2^8$ for K-Distributed) and because K-Replicated's design involves more descents. This enables K-Replicated to perform more exploration, at the cost of a much greater budget.

4.4 Impact of the population size

In this section, we analyze the effect of the population size on K-Distributed's performance. In Figure 9, we show the convergence profiles for each distinct population size of K-Distributed on two illustrative BBOB functions. We can see that the value of the population size allowing to reach the fastest a function quality depends both on the considerd function and the considered quality. For an easy target quality, or for a function with a very simple shape (such as the sphere

function f_1), the descents underlying K-Distributed are clearly ordered by K value (hence by population size) : the $K = 2^0$ descent is better than the $K = 2^1$ one, which is better than the $K = 2^2$ one, and so forth. However, for a more complex function (such as f_{17}) and for more challenging target qualities, some descents may stop being competitive after a given time, ordered by the value of K. We also see that the time before a descent stops being effective highly depends on the underlying population size. To support these observations, we report in Table 4 the average $log_2 K$ value of the first descent to find a solution for the considered functions and targets. We clearly see that lower population sizes are better suited in the first stages of the search (i.e. for the columns with the highest power of 10). However, for relatively difficult targets the best population size varies substantially, with $log_2 K$ ranging from 0.1 to 7.5. Notice that these targets correspond to the highest speedups of K-Distributed over sequential IPOP-CMA-ES. Besides, the best population size highly depends on the considered function, and no population size is inherently better than another. Consequently, since we lack a reliable way to predict which population size is more effective, the best strategy is to give an equal chance to each size and start them all at the beginning of the execution, which is precisely how K-Distributed operates. Moreover, let us emphasize that in the design of K-Distributed parallel evaluation strategy, the number of cores used is proportional to the population size. This makes the duration of the iterations closer among the different population sizes. As a result, the convergence of each descent operates on a more similar time scale. Without such a parallel evaluation, the descent with larger population sizes would require much more time and would be less competitive compared to the ones with lower population sizes.

5 Experimental results on a local cluster

The results discussed in the previous section were obtained using the supercomputer Fugaku. This could raise the question of whether the conclusions drawn can be generalized to more accessible computing platforms, which might constitute an issue from a reproducibility and HPC perspective when assessing the performance of a parallel algorithm. This is precisely the goal of this section, where we consider running the different algorithms using a relatively smaller local cluster. Specifically, we use the *gros* cluster of the Grid'5000 platform[7] equipped with 124 nodes, each node having one Intel Xeon Gold 5220 CPU with 18 cores. In order to make our experiments manageable, and due to the restrictions imposed by internal usage rules of the Grid'5000 platform, we fix the time limit to 1 hour for all experimented algorithms. The parameters underlying our initial experimental set-up (including the benchmark functions, the additional cost controlling function evaluation granularity, and the values of λ_{start} and $K_{\max}$) remain unchanged. But since the number of available CPU cores is much lower here than on Fugaku, we cannot process each evaluation in parallel on a dedicated CPU core. Therefore, we use only one thread per MPI process, and at each iteration of CMA-ES each thread processes $\lambda_{start} = 12$ evaluations in serial on one CPU core. This enables us to consider 12 time less CPU cores, hence up to 512 using 29 nodes. Although this setup may not yield the maximum possible speedups, it provides a fair basis for evaluating and analyzing the relative performance of the parallel techniques and strategies under consideration within a smaller, more typical HPC environment. Due to space constraints, we only focus in the following on how our previous key findings translate with respect to: (i) the introduced BLAS/LAPACK optimizations, and (ii) the proposed parallel evaluation strategies.

Firstly, we report in Figure 10 the performance gain obtained by the BLAS and LAPACK operations as discussed previously in Section 3.1.2, and subsequently analyzed in Figure 5d on

[7]See: `https://www.grid5000.fr`

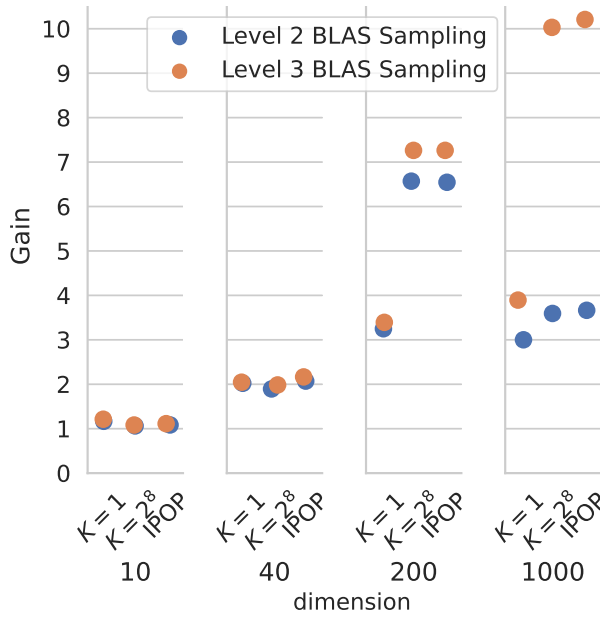
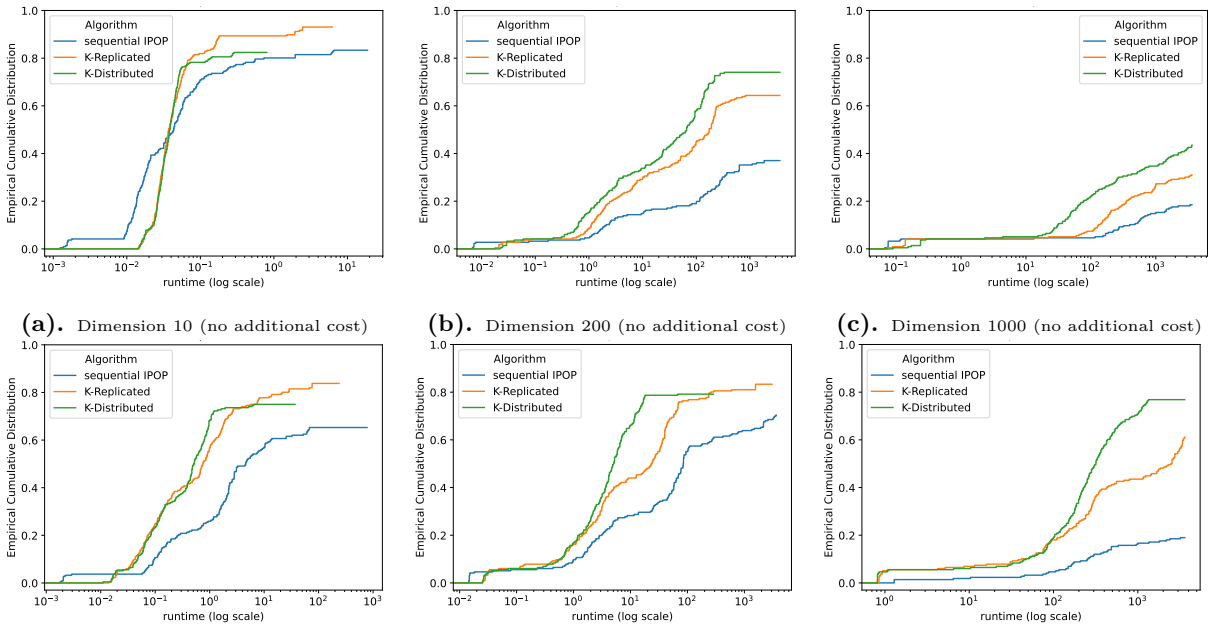


Figure 10: Performance gains on one Xeon core over the reference C code (without BLAS and LAPACK) for all the linear algebra part, with LAPACK for the eigendecomposition and Level 3 BLAS for the C matrix adaptation, when using Level 2 or Level 3 BLAS routines for the sampling.

Fugaku. From Figure 10, it is clear that the BLAS and LAPACK optimizations introduced in the CMA-ES reference C code are valid when implemented and executed on one Xeon core of the considered local cluster. Besides, we can see that the gains offered by BLAS and LAPACK for the CMA-ES algebraic operations are even better compared to the ones obtained on Fugaku. In fact, we here obtain up to $\times 10$ gain against solely $\times 2.5$ gain previously. Let us emphasize that we did not customize our code in anyway on such an environment, but we simply relied on the standard Intel oneAPI Math Kernel Library to run our code properly on this architecture.

Secondly, we report in Figure 11 the ECDF profile obtained by the different experimented algorithms. As discussed previously, the ECDF allows us to provide a fair idea on the relative performance of the different algorithms independently of the considered functions or quality targets. This figure is to be contrasted with Figure 8 reported previously for the supercomputer Fugaku. We can see that, with few exceptions, the results from Figure 11 are fully consistent with the ones obtained on Fugaku: For relevant function computation grains or dimensions, the ECDF curves indicate that K-Distributed is overall better than K-Replicated, which is better than sequential IPOP-CMA-ES, demonstrating that the parallel evaluation strategy implied by the designed K-Distributed is the most accurate when parallelizing IPOP-CMA-ES. For completeness, we can notice few changes in the relative performance of the parallel algorithms, which happen only for the function settings having the smallest computation grain, namely dimension 10 and dimension 40 without additional function evaluation cost. In particular, we can see that in the latter situation, the gap between K-Distributed and K-Replicated is less pronounced relatively to the Fugaku experiments. We believe that this is to be attributed to the difference in performance between the high performance TofuD [4] Fugaku interconnect, and the standard Ethernet network underlying the considered Grid'5000 local cluster. More precisely, the former offers a 6.25 GB/s maximum theoretical throughput against a 40.8 GB/s one in the Fugaku environment. For low computation grain or dimension, the low K-value descents (more numerous in K-replicated) are indeed less communication sensitive than the descents with greater K-values. Notice however that the relative gap in performance between



(a). Dimension 10 (no additional cost) **(b).** Dimension 200 (no additional cost) **(c).** Dimension 1000 (no additional cost)

(d). Dimension 40 (no additional cost) **(e).** Dimension 40, additional cost of 1ms **(f).** Dimension 40, additional cost of 100ms

Figure 11: ECDFs, obtained on the local cluster, of each algorithm for various dimensions and granularities.

the two parallel strategies becomes significant as soon as slightly higher function evaluation grains or dimensions are experimented, which demonstrate the relevancy of our parallel design. Lastly, we found that the speedups obtained on the local cluster are lower than those obtained on Fugaku. This is without surprise since in the local cluster setting, our experiments employ only 512 CPU cores (against 6144 CPU cores on Fugaku), and 12 function evaluations are performed by each CPU core (while these 12 function evaluations were performed in parallel on 12 cores on Fugaku). Notice however that we found that the overall parallel efficiencies (that is the ratios between the achieved speedups and the number of cores) remain similar in the two experiments.

To summarize, this additional set of experiments demonstrates that our techniques are not limited to deployment on large-scale supercomputers such as Fugaku. They remain effective and worth considering even when only a more modest local cluster is available. This also highlights the portability and adaptability of the proposed methods, making them accessible to a broader range of end-users, especially from the optimization community. Moreover, the consistency of the observed performance trends across different hardware platforms reinforces the robustness of our approach and its practical relevance in real-world HPC scenarios.

6 Conclusion

In this paper, we investigated two parallel strategies for the IPOP-CMA-ES (Covariance Matrix Adaptation Evolution Strategy with Increasing Population) algorithm, designed for large black-box optimization problems on thousands of CPU cores. Both strategies leveraged BLAS and LAPACK routines to accelerate linear algebra operations, which required the rewriting of some operations to successfully benefit from the more efficient Level 3 BLAS. The first approach, K-Replicated, performs multiple descents with identical population sizes, increasing the population size as descents conclude and thus mirroring the progression of IPOP-CMA-ES. On the

other hand the second strategy, K-Distributed, adapts IPOP-CMA-ES differently by initiating all descents simultaneously, each with a distinct population size.

Thanks to experiments on the supercomputer Fugaku, using MPI+OpenMP implementations on 128 A64FX CPUs of 48 cores each (6144 cores in total), with a reference blackbox optimization benchmark extended with coarser computation grains, we showed that these parallel strategies greatly improved on the convergence speed of the original sequential IPOP-CMA-ES, reaching speedups up to several thousand. Notably, K-Distributed outperformed K-Replicated in the vast majority of cases. Moreover, due to its concurrent processing of multiple descents with distinct population sizes, K-Distributed occasionally exhibited super-linear speedups up to 18080× on 6144 cores. We complemented these results with a detailed analysis of the superior performance of K-Distributed. According to our results, if one has a given allocated time on a large-scale parallel architecture, we recommend running K-Distributed and possibly restarting each descent once finished until the time is up. Moreover, additional experiments conducted on a local cluster with 512 cores demonstrate that the proposed techniques are not specific to Fugaku, highlighting their relevance and applicability across a broader range of HPC environments.

This study opens up several interesting research avenues. The proposed large-scale techniques could serve as a basis for parallelizing other CMA-ES variants, such as the large-scale ones[38, 41, 40] Additionally, they could be integrated with other optimization heuristics, such as global optimization [53] or hyper-parameter tuning [42]. Furthermore, investigating methods to predict the most effective CMA-ES population size for an objective function, either beforehand or at runtime, could strongly benefit to both sequential and parallel blackbox optimization.

7 Acknowledgments

The large-scale experiments presented in this paper were carried out using the supercomputer Fugaku, provided by the RIKEN Center for Computational Science. The experiments conducted on the local cluster were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`). The authors thank the Hauts-de-France region for partly funding David Redon's PhD thesis.

References

[1] Graph500 bfs list, june 2024. `https://graph500.org/?page_id=1285`, 2024.

[2] Hpcg list, june 2024. `https://www.top500.org/lists/hpcg/2024/06/`, 2024.

[3] Top500 list, june 2024. `https://www.top500.org/lists/top500/2024/06/`, 2024.

[4] Yuichiro Ajima, Takahiro Kawashima, Takayuki Okamoto, Naoyuki Shida, Kouichi Hirai, Toshiyuki Shimizu, Shinya Hiramoto, Yoshiro Ikeda, Takahide Yoshikawa, Kenji Uchida, et al. The tofu interconnect d. 2018 IEEE International Conference on Cluster Computing (CLUSTER), pages 646–654. IEEE, 2018.

[5] Vahab Akbarzadeh, Albert Hung-Ren Ko, Christian Gagné, and Marc Parizeau. Topography-aware sensor deployment optimization with cma-es. Parallel Problem Solving from Nature, PPSN XI: 11th International Conference, Kraków, Poland, September 11-15, 2010, Proceedings, Part II 11, pages 141–150. Springer, 2010.

[6] Youhei Akimoto and Nikolaus Hansen. Projection-based restricted covariance matrix adaptation for high dimension. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2016, pages 197–204, 2016.

[7] Viktor Arkhipov, Maxim Buzdalov, and Anatoly Shalyto. An asynchronous implementation of the limited memory cma-es. 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), pages 707–712. IEEE, 2015.

[8] Asma Atamna. Benchmarking ipop-cma-es-tpa and ipop-cma-es-msr on the bbob noiseless testbed. Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO), pages 1135–1142, 2015.

[9] Anne Auger and Nikolaus Hansen. Performance evaluation of an advanced local search evolutionary algorithm. 2005 IEEE congress on evolutionary computation (CEC), pages Vol. 2: 1777–1784. IEEE, 2005.

[10] Anne Auger and Nikolaus Hansen. A restart cma evolution strategy with increasing population size. 2005 IEEE Congress on Evolutionary Computation (CEC), pages Vol. 2: 1769–1776. IEEE, 2005.

[11] Nacim Belkhir, Johann Dréo, Pierre Savéant, and Marc Schoenauer. Parameter setting for multicore cma-es with large populations. Artificial Evolution: 12th International Conference, Evolution Artificielle, EA 2015, Lyon, France, October 26-28, 2015. Revised Selected Papers 12, pages 109–122. Springer, 2016.

[12] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. International conference on machine learning (ICML), pages 115–123. PMLR, 2013.

[13] Huangke Chen, Ran Cheng, Jinming Wen, Haifeng Li, and Jian Weng. Solving large-scale many-objective optimization problems by covariance matrix adaptation evolution strategy with scalable small subpopulations. *Informatics and Computer Science Intelligent Systems Applications (Information Sciences)*, 509:457–469, 2020.

[14] Ivo Couckuyt, Frederick Declercq, Tom Dhaene, Hendrik Rogier, and Luc Knockaert. Surrogate-based infill optimization applied to electromagnetic problems. *International Journal of RF and Microwave Computer-Aided Engineering*, 20(5):492–501, 2010.

[15] Qiqi Duan, Guochen Zhou, Chang Shao, Yijun Yang, and Yuhui Shi. Collective learning of low-memory matrix adaptation for large-scale black-box optimization. Parallel Problem Solving from Nature–PPSN XVII: 17th International Conference, PPSN 2022, Dortmund, Germany, September 10–14, 2022, Proceedings, Part II, pages 281–294. Springer, 2022.

[16] Qiqi Duan, Guochen Zhou, Chang Shao, Yijun Yang, and Yuhui Shi. Distributed evolution strategies for large-scale optimization. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) Companion, pages 395–398, 2022.

[17] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[18] Ahmed S Elshall, Hai V Pham, Frank T-C Tsai, Le Yan, and Ming Ye. Parallel inverse modeling and uncertainty quantification for computationally demanding groundwater-flow models using covariance matrix adaptation. *Journal of Hydrologic Engineering (JHE)*, 20(8):04014087, 2015.

[19] Hongbing Fang, Kiran Solanki, and MF Horstemeyer. Numerical simulations of multiple vehicle crashes and multidisciplinary crashworthiness optimization. *International Journal of Crashworthiness*, 10(2):161–172, 2005.

[20] Garuda Fujii, Youhei Akimoto, and Masayuki Takahashi. Exploring optimal topology of thermal cloaks by cma-es. *Applied Physics Letters (APL)*, 112(6), 2018.

[21] Garuda Fujii, Masayuki Takahashi, and Youhei Akimoto. Cma-es-based structural topology optimization using a level set boundary expression—application to optical and carpet cloaks. *Computer Methods in Applied Mechanics and Engineering (CMAME)*, 332:624–643, 2018.

[22] Myeong-Seok Go, Jae Hyuk Lim, and Seungchul Lee. Physics-informed neural network-based surrogate model for a virtual thermal sensor with real-time simulation. *International Journal of Heat and Mass Transfer*, 214:124392, 2023.

[23] Doug Hakkarinen, Tracy Camp, Zizhong Chen, and Allan Haas. Reduced data communication for parallel cma-es for reacts. 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pages 97–101. IEEE, 2012.

[24] Nikolaus Hansen, Anne Auger, Steffen Finck, and Raymond Ros. Real-Parameter Black-Box Optimization Benchmarking 2009: Experimental Setup. Research Report RR-6828, INRIA, 2009.

[25] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions. Research Report RR-6829, INRIA, 2009.

[26] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation (ECJ)*, 11(1):1–18, 2003.

[27] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[28] Nikolaus Hansen, Tea Tusar, Olaf Mersmann, Anne Auger, and Dimo Brockhoff. Coco: The experimental procedure. *arXiv preprint arXiv:1603.08776*, 2016.

[29] Martina Hasenjäger, Bernhard Sendhoff, Toyotaka Sonoda, and Toshiyuki Arima. Three dimensional evolutionary aerodynamic design optimization with cma-es. Proceedings of the 7th annual conference on Genetic and evolutionary computation (GECCO), pages 2173–2180, 2005.

[30] Xiaoyu He, Zibin Zheng, and Yuren Zhou. Mmes: Mixture model-based evolution strategy for large-scale optimization. *IEEE Transactions on Evolutionary Computation (TEVC)*, 25(2):320–333, 2020.

[31] Jaebak Hwang, Sungahn Ko, and Tsz-Chiu Au. Calibrating dynamic traffic assignment models by parallel search using active-cma-es. 2021 IEEE International Intelligent Transportation Systems Conference (ITSC), pages 3265–3270. IEEE, 2021.

[32] A Jahangirian and A Shahrokhi. Aerodynamic shape optimization using efficient evolutionary algorithms and unstructured cfd solver. *Computers & Fluids*, 46(1):270–276, 2011.

[33] Jin Jin, Chuan Yang, and Yi Zhang. An improved cma-es for solving large scale optimization problem. Advances in Swarm Intelligence: 11th International Conference, ICSI 2020, Belgrade, Serbia, July 14–20, 2020, Proceedings 11, pages 386–396. Springer, 2020.

[34] Jérôme Henri Kämpf and Darren Robinson. A hybrid cma-es and hde optimisation algorithm with application to solar energy potential. *Applied Soft Computing*, 9(2):738–745, 2009.

[35] Muhammad Firmansyah Kasim, Duncan Watson-Parris, Lucia Deaconu, Sophy Oliver, P Hatfield, Dustin H Froula, Gianluca Gregori, Matt Jarvis, Samar Khatiwala, Jun Korenaga, et al. Building high accuracy emulators for scientific simulations with deep neural architecture search. *Machine Learning: Science and Technology*, 3(1):015013, 2021.

[36] Jeffrey Larson, Matt Menickelly, and Stefan M Wild. Derivative-free optimization methods. *Acta Numerica*, 28:287–404, 2019.

[37] François Lemaire and Louis Roussel. Parameter estimation using integral equations. *Maple Transactions*, 4(1), Jun. 2024.

[38] Zhenhua Li and Qingfu Zhang. A simple yet efficient evolution strategy for large-scale black-box optimization. *IEEE Transactions on Evolutionary Computation (TEVC)*, 22(5):637–646, 2017.

[39] Zhenhua Li, Qingfu Zhang, Xi Lin, and Hui-Ling Zhen. Fast covariance matrix adaptation for large-scale black-box optimization. *IEEE transactions on cybernetics*, 50(5):2073–2083, 2018.

[40] Ilya Loshchilov. A computationally efficient limited memory cma-es for large scale optimization. Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO), pages 397–404, 2014.

[41] Ilya Loshchilov, Tobias Glasmachers, and Hans-Georg Beyer. Large scale black-box optimization by limited-memory matrix adaptation. *IEEE Transactions on Evolutionary Computation (TEVC)*, 23(2):353–358, 2018.

[42] Ilya Loshchilov and Frank Hutter. Cma-es for hyperparameter optimization of deep neural networks. *International Conference on Learning Representations (ICLR). Workshop Track*, 2016.

[43] Ilya Loshchilov, Marc Schoenauer, and Michele Sebag. Alternative restart strategies for cma-es. International Conference on Parallel Problem Solving from Nature (PPSN), pages 296–305. Springer, 2012.

[44] Atsuo Maki, Youhei Akimoto, and Umeda Naoya. Application of optimal control theory based on the evolution strategy (cma-es) to automatic berthing (part: 2). *Journal of Marine Science and Technology (JMST)*, 26:835–845, 2021.

[45] Atsuo Maki, Naoki Sakamoto, Youhei Akimoto, Hiroyuki Nishikawa, and Naoya Umeda. Application of optimal control theory based on the evolution strategy (cma-es) to automatic berthing. *Journal of Marine Science and Technology (JMST)*, 25:221–233, 2020.

[46] René Meier, Martin Pippel, Frank Brandt, Wolfgang Sippl, and Carsten Baldauf. Paradocks: a framework for molecular docking with population-based metaheuristics. *Journal of chemical information and modeling*, 50(5):879–889, 2010.

[47] Message Passing Interface Forum. MPI: A message-passing interface standard, version 4.1. Available: `https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf`, 2023.

[48] Jorge J Moré and Stefan M Wild. Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1):172–191, 2009.

[49] Christian L Müeller and Ivo F Sbalzarini. A tunable real-world multi-funnel benchmark problem for evolutionary optimization-and why parallel island models might remedy the failure of cma-es on it. International Conference on Evolutionary Computation, pages Vol. 2: 248–253. SCITEPRESS, 2009.

[50] Pradeep Mugunthan, Christine A Shoemaker, and Rommel G Regis. Comparison of function approximation, heuristic, and derivative-based methods for automatic calibration of computationally expensive groundwater bioremediation models. *Water Resources Research*, 41(11), 2005.

[51] Christian L Müller, Benedikt Baumgartner, Georg Ofenbeck, Birte Schrader, and Ivo F Sbalzarini. pcmalib: a parallel fortran 90 library for the evolution strategy with covariance matrix adaptation. Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO), pages 1411–1418. ACM, 2009.

[52] Christian L Muller, Benedikt Baumgartner, and Ivo F Sbalzarini. Particle swarm cma evolution strategy for the optimization of multi-funnel landscapes. 2009 IEEE Congress on Evolutionary Computation (CEC), pages 2685–2692. IEEE, 2009.

[53] Rémi Munos. Optimistic optimization of a deterministic function without the knowledge of its smoothness. 25th Annual Conference on Neural Information Processing Systems (NIPS), pages 783–791, 2011.

[54] Daniele Muraro and Rui Dilão. A parallel multi-objective optimization algorithm for the calibration of mathematical models. *Swarm and Evolutionary computation*, 8:13–25, 2013.

[55] Yuichi Nagata. The lens design using the cma-es algorithm. Genetic and Evolutionary Computation Conference (GECCO), pages 1189–1200. Springer, 2004.

[56] Ryohei Okazaki, Takekazu Tabata, Sota Sakashita, Kenichi Kitamura, Noriko Takagi, Hideki Sakata, Takeshi Ishibashi, Takeo Nakamura, and Yuichiro Ajima. Supercomputer fugaku cpu a64fx realizing high performance, high-density packaging, and low power consumption. *Fujitsu Technical Review*, pages 2020–03, 2020.

[57] Mohammad Nabi Omidvar and Xiaodong Li. A comparative study of cma-es on large scale global optimisation. Australasian Joint Conference on Artificial Intelligence (AJCAI), pages 303–312. Springer, 2010.

[58] OpenMP Architecture Review Board. OpenMP Application Program Interface, version 5.2. Available: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`, 2021.

[59] S Surender Reddy, BK Panigrahi, Rupam Kundu, Rohan Mukherjee, and Shantanab Debchoudhury. Energy and spinning reserve scheduling for a wind-thermal power system using cma-es with mean learning technique. *International Journal of Electrical Power & Energy Systems (JEPE)*, 53:113–122, 2013.

[60] Raymond Ros and Nikolaus Hansen. A simple modification in cma-es achieving linear time and space complexity. International conference on parallel problem solving from nature (PPSN), pages 296–305. Springer, 2008.

[61] Thomas Philip Runarsson. Asynchronous parallel (1+ 1)-cma-es for constrained global optimisation. IJCCI (ECTA), pages 266–272, 2014.

[62] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, et al. Co-Design for A64FX Manycore Processor and "Fugaku". SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–15. IEEE, 2020.

[63] Hannes Schwarz, Lars Kotthoff, Holger Hoos, Wolf Fichtner, and Valentin Bertsch. Improving the computational efficiency of stochastic programs using automated algorithm configuration: an application to decentralized energy systems. *Annals of Operations Research*, pages 1–22, 2019.

[64] Urban Škvorc, Tome Eftimov, and Peter Korošec. Gecco black-box optimization competitions: progress from 2009 to 2018. Proceedings of the genetic and evolutionary computation conference (GECCO) companion, pages 275–276, 2019.

[65] Hassan Smaoui, Lahcen Zouhri, Sami Kaidi, and Erick Carlier. Combination of fem and cma-es algorithm for transmissivity identification in aquifer systems. *Hydrological Processes*, 32(2):264–277, 2018.

[66] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. *Advances in neural information processing systems (NIPS)*, 26, 2013.

[67] Ryoji Tanabe and Alex Fukunaga. Tuning differential evolution for cheap, medium, and expensive computational budgets. 2015 IEEE Congress on Evolutionary Computation (CEC), pages 2018–2025. IEEE, 2015.

[68] André Thomaser, Anna V Kononova, Marc-Eric Vogt, and Thomas Bäck. One-shot optimization for vehicle dynamics control systems: towards benchmarking and exploratory landscape analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 2036–2045, 2022.

[69] Konstantinos Varelas. Benchmarking large scale variants of cma-es and l-bfgs-b on the bbob-largescale testbed. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) Companion, pages 1937–1945, 2019.

[70] Konstantinos Varelas, Anne Auger, Dimo Brockhoff, Nikolaus Hansen, Ouassim Ait ElHara, Yann Semet, Rami Kassab, and Frédéric Barbaresco. A comparative study of large-scale variants of cma-es. Parallel Problem Solving from Nature–PPSN XV: 15th International Conference, Coimbra, Portugal, September 8–12, 2018, Proceedings, Part I 15, pages 3–15. Springer, 2018.

[71] Tian Wang, Mingqi Shao, Rong Guo, Fei Tao, Gang Zhang, Hichem Snoussi, and Xingling Tang. Surrogate model via artificial intelligence method for accelerating screening materials and performance prediction. *Advanced Functional Materials*, 31(8):2006245, 2021.

[72] Kensuke Watanabe, Takafumi Nose, Kiyofumi Suzuki, and Shuichi Chiba. Application development environment for supercomputer fugaku. `https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article07.html`, 2020.

[73] John H Wilkinson and Christian Reinsch. *Handbook for Automatic Computation: Volume II: Linear Algebra*, volume 186. Springer Science & Business Media, 2012.