UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

eXascale Infolab

UNIVERSITY OF FRIBOURG

MASTER THESIS

Study and Extension of the DiBB Framework for Distributing Black-Box Optimization

Author:
Luca Sven Rolshoven

Supervisor:
Prof. Dr. Philippe Cudré-Mauroux

Co-Supervisor:
Dr. Giuseppe Cuccu

September 03, 2021

eXascale Infolab
Department of Informatics

Boulevard de Pérolles 90 • 1700 Fribourg • Switzerland
phone +41 (26) 300 84 65 • diuf-secr@unifr.ch • www3.unifr.ch/inf

Acknowledgements

First of all, I want to thank my supervisor Dr. Giuseppe Cuccu, for his outstanding support and his valuable advice, both for the development of this thesis as well as my personal life and career. I also want to thank my second supervisor Prof. Dr. Philippe Cudré-Mauroux for letting me do the thesis at the eXascale Infolab, and for being an excellent professor. Your classes were genuinely interesting, and I truly enjoyed them. I would like to extend my sincere thanks to Prof. Dr. Tobias Glasmachers for being part of this exciting project and for his extensive knowledge.

My thanks also go to Fabien Vorpe, who did his thesis at the same time as me. I profoundly enjoyed our meetings and discussions together with Dr. Giuseppe Cuccu.

Finally, and most importantly, I want to thank my family and my girlfriend Ana for their love and support, not only throughout this thesis but my entire master's studies. Without you, I would not be where I am today.

Abstract

Luca Sven Rolshoven

Study and Extension of the DiBB Framework for Distributing Black-Box Optimization

Black-Box Optimization (BBO) has been applied to many problems including Reinforcement Learning (RL), Control Problems, and parameter optimization in general. Since classical optimization algorithms rely on additional information about the structure of a problem, they can only tackle problems which comply with these requirements. BBO algorithms consider the problem to be a black box, and thus they do not have any requirement about their structure, which means that they can be used on a wide range of problems. Unfortunately, most BBO algorithms have a high time complexity, which is often caused by the update of a covariance matrix and makes optimizing problems with many parameters infeasible. To that end, it has been proposed to use a diagonal covariance matrix, which leads to a linear complexity of the updates. However, the performance of this approach is often poor because all of the correlations between the variables are discarded. To solve this problem, another method called BD-NES was proposed that uses a block-diagonal covariance matrix, thereby reducing the computational complexity of the family of Natural Evolution Strategies (NES) from $\mathcal{O}(d^3)$ to $\mathcal{O}(b^3)$, where d is the dimensionality of the problem, and b is the size of the largest block in the covariance matrix. This method allows to tune the trade-off between wall-clock speed and sample efficiency of the optimization procedure. DiBB is a framework for distributed BBO that has been created to leverage this property. It generalizes the concept behind BD-NES to any BBO algorithm, and it distributes the optimization of the blocks. The goal of this thesis is to extend DiBB and to study its performance. To do so, I ran several experiments using DiBB with CMA-ES on the BBOB and the BBOB-Largescale suite of the COCO benchmark. The results show that DiBB is a competitive approach for tackling larger problems with BBO. DiBB with CMA-ES outperforms the separable version of CMA-ES with a diagonal covariance matrix and it is faster than then standard CMA-ES with a full covariance matrix. Using DiBB, it is possible to optimize problems that were infeasible for BBO previously. Future research may focus on heuristics for choosing the blocks and their sizes as well as the interaction between different kinds of BBO algorithms and DiBB.

Keywords: black-box optimization, distributed computing, evolution strategies, framework, parameter optimization

Contents

Abstract **v**

1 Introduction **1**
1.1 Black-Box Optimization . . . 1
1.1.1 Evolution Strategies . . . 4
1.1.2 Covariance Matrix Adaption Evolution Strategy (CMA-ES) . . . 4
1.1.3 Hessian Evolution Strategy (HE-ES) . . . 6
1.1.4 Block Diagonal Natural Evolution Strategy (BD-NES) . . . 7
1.2 COCO: Comparing Continuous Optimizers . . . 8
1.2.1 BBOB . . . 8
1.2.2 BBOB-Largescale . . . 8
1.2.3 Past Results . . . 9
1.2.4 General Usage . . . 9
1.2.5 Inner Workings . . . 9
1.2.6 Data Format . . . 10
1.2.7 Understanding the Plots . . . 12
1.3 Distributed Computing . . . 12
1.3.1 Ray . . . 14
1.4 Contributions . . . 16

2 Method **19**
2.1 From BD-NES to Distributed Black-Box Optimization . . . 19
2.2 Using Ray for DiBB . . . 19
2.3 Cluster Setup . . . 20
2.4 Inner Structure of DiBB . . . 21
2.5 DiBB Usage . . . 22
2.5.1 Initializing DiBB . . . 22
2.5.2 Running DiBB . . . 24
2.5.3 Resetting DiBB . . . 26
2.5.4 Writing Custom Hooks . . . 27

3 Experiments **29**
3.1 Research Questions . . . 29
3.2 Experimental Setup . . . 29
3.2.1 Cluster Setup . . . 29
3.2.2 Data Merging . . . 30
3.3 Experiments . . . 32
3.4 Results . . . 35
3.4.1 Experiment 1 . . . 35
3.4.2 Experiment 2 . . . 37
3.4.3 Experiment 3 . . . 41
3.4.4 Experiment 4 . . . 45
3.4.5 Comparing DiBB to Other Methods . . . 49

3.4.6 Timings 54

4 Discussion and Conclusion **61**
4.1 Future Work 63

Bibliography **65**

List of Figures

1.1 Rastrigin function . . . 3
1.2 Example ECDF runtime graph . . . 13

2.1 Illustration of the workflow inside DiBB . . . 23

3.1 ECDF plot for experiment `bbob_fnb` . . . 37
3.2 Plots of functions f1 to f8 in the BBOB suite that were optimized with DiBB and CMA-ES using a constant number of blocks. . . . 38
3.3 Plots of functions f9 to f16 in the BBOB suite that were optimized with DiBB and CMA-ES using a constant number of blocks. . . . 39
3.4 Plots of functions f17 to f24 in the BBOB suite that were optimized with DiBB and CMA-ES using a constant number of blocks. . . . 40
3.5 Comparison of the ECDF plots over all 24 functions of experiments `bbob_fnb` (left) and `bbob_fbs` (right). . . . 41
3.6 Plots of functions f1 to f8 in the BBOB suite that were optimized with DiBB and CMA-ES using a constant block size. . . . 42
3.7 Plots of functions f9 to f16 in the BBOB suite that were optimized with DiBB and CMA-ES using a constant block size. . . . 43
3.8 Plots of functions f17 to f24 in the BBOB suite that were optimized with DiBB and CMA-ES using a constant block size. . . . 44
3.9 ECDF plot for the experiment with a fixed number of blocks on the BBOB-Largescale suite . . . 45
3.10 Plots of functions f1 to f8 in the BBOB-Largescale suite that were optimized with DiBB and CMA-ES using a constant number of blocks. . . . 46
3.11 Plots of functions f9 to 16 in the BBOB-Largescale suite that were optimized with DiBB and CMA-ES using a constant number of blocks. . . . 47
3.12 Plots of functions f17 to f24 in the BBOB-Largescale suite that were optimized with DiBB and CMA-ES using a constant number of blocks. 48
3.13 ECDF plot for the experiment with a fixed number of blocks on the BBOB-Largescale suite . . . 50
3.14 Plots of functions f1 to f8 in the BBOB-Largescale suite that were optimized with DiBB and CMA-ES using a constant block size. . . . 51
3.15 Plots of functions f9 to f16 in the BBOB-Largescale suite that were optimized with DiBB and CMA-ES using a constant block size. . . . 52
3.16 Plots of functions f17 to f24 in the BBOB-Largescale suite that were optimized with DiBB and CMA-ES using a constant block size. . . . 53
3.17 Comparison between CMA-ES baseline, DiBB and CMA-ES with diagonal covariance matrix . . . 56
3.18 Comparison of CMA-ES, DiBB with CMA-ES and sepCMA on functions f1 to f8 on the BBOB-Largescale suite. . . . 57
3.19 Comparison of CMA-ES, DiBB with CMA-ES and sepCMA on functions f9 to f16 on the BBOB-Largescale suite. . . . 58

3.20 Comparison of CMA-ES, DiBB with CMA-ES and sepCMA on functions f17 to f24 on the BBOB-Largescale suite. 59
3.21 Durations of the experiment with a fixed block size on the BBOB-Largescale 60
3.22 Wall-clock time for the optimization of problems with dimensions 80, 160, and 320 for two different number of blocks each. 60

Chapter 1

Introduction

The aim of this thesis is to explore the capabilities of DiBB, a novel framework for black-box optimization that is introduced in Chapter 2.1. In this chapter, I introduce the basic concepts that are needed for the understanding of this work. First, the concept of black-box optimization is introduced, and three different Evolution Strategies are presented. Afterward, I present the COCO benchmark, which is used in the experiments of this thesis. To conclude this chapter, I give a brief introduction to distributed computing.

1.1 Black-Box Optimization

A black box is a fictional representation of a concrete system that has observable inputs and outputs, but the interior structure of the system is either irrelevant or not known (Bunge, 1963). In the context of optimization, this means that we do not have any information about the structure of the problem that needs to be solved. The properties of black-box optimization are best described by contrasting them with the ones from classical optimization, where we take into account also the structure of the problem at hand.

The goal of optimization is to find the best solution for a specific problem. Thus, given an objective function $f(x)$, the aim is to find the optimal point x^* such that $f(x^*)$ is a global optimum. Depending on the task, this optimum might be a maximum or a minimum. Additionally, some optimization tasks also require the solution to fulfill certain conditions, which can be expressed in the form of equality and inequality constraints. The solution of such a problem is not the global optimum of the objective function over the entire domain but rather the optimum within the feasible region, i.e. the set of points that complies with the constraints that were provided. In the context of black-box optimization, the objective function is often referred to as *fitness function*.

Note that it is important to know whether we are talking about a maximization or minimization problem. In the latter case, a lower value of the fitness function would correspond to a better solution, which may sound counter-intuitive when thinking about the phrase *survival of the fittest*. Therefore, it is important to remember that the fittest individual, i.e. the best solution, is not always the one with the highest fitness score.

Most classical optimization methods are based on the *exploitation* of first- and second-order information about a problem, i.e. they make use of the gradient and sometimes also the Hessian matrix of the fitness function to gain a better understanding about the optimization landscape. Consider a minimization problem with a simple quadratic function f in one variable x, e.g. $f(x) = 2x^2 + 4x + 1$. The optimum of this problem can be found analytically by setting the first derivative to zero. In this example, we get $f'(x) = 4x + 4$ and $f'(x^*) = 0 \Rightarrow x^* = -1$. Hence, the

optimum is given by $f(-1) = -1$. Clearly, this is just a special case, and usually optimization problems are much harder. However, it serves its purpose as an example to demonstrate how first-order information can be used to find the optimum of a function. Of course there are also many algorithms in classical optimization that tackle harder problems, e.g. high-dimensional non-convex functions with additional constraints. But irrespective of the complexity of these algorithms, they all use information about the structure of the fitness landscape during optimization.

If such information is available, it is of course valid to use it. However, there are some disadvantages. The term *mode* usually refers to the maximum of the function. However, in this thesis I will use the term as defined in the Mathematical Programming Glossary (Sauppe, 2006–18), where it can be either a maximum or a minimum, i.e. an optimum in general. A function that has only one mode is called *unimodal*. Unimodality is strongly related to convexity and concavity of a function. On the other hand we have *multimodal* problems, which make up the majority of problems. These problems often have many local optima. Because classical optimization methods make use of first- and second-order information about the fitness function, such local optima are attractors for these procedures. Often, the algorithms are unable to leave the basin of attraction of a local optimum and thus converge prematurely, i.e. their search becomes limited to this basin of attraction and the local minimum or maximum is eventually returned. This happens often for *deceptive problems*, where the landscape of the fitness function does not provide useful information about how to reach the global optimum (Lehman, 2012; Conti et al., 2018; Mousavirad, Bidgoli, and Rahnamayan, 2019).

One such problem is the Rastrigin function. It was first proposed in 1974 (Rastrigin, 1974) as a two-dimensional problem, just as depicted in Figure 1.1. It is easy to see that there are many local optima and that it would be hard to find the global optimum if we were to start at a random location and then only follow the direction of the gradient at that point. In 1990, Rudolph generalized the concept of the Rastrigin function to arbitrary dimensions in his PhD thesis (Rudolph, 1990). The resulting function is given by

$$f(x) = An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(\omega x_i) \right], \tag{1.1}$$

where n is the dimensionality of the problem. The parameters A and ω are usually chosen to be $A = 10$ and $\omega = \pi$, and the function is usually evaluated for vectors x with $x_i \in [-5.12, 5.12]$. With an increasing number of dimensions, the number of local minima is also increased, making it harder to solve the problem with classical optimization. This is the reason why a high-dimensional Rastrigin function is hard to optimize with classical optimization methods. On the other hand, it is often used to assess the performance of black-box algorithms.

While black-box optimization algorithms can also converge to local optima, they are typically better suited for multimodal problems because they are *multi-agent* algorithms. By using a set of candidate solutions rather than looking at one possible solution at a time, it is easier for the optimization method to escape local optima. The reason behind this is that the individual solutions that are stuck in a local optimum will have a worse fitness score than the ones that got past the local optimum. This is referred to as *exploration*, and it is a key strength of black-box optimization algorithms.

Classical optimization methods are good at exploiting differential information, but they are bad at exploring different ares of the fitness landscape. An example of

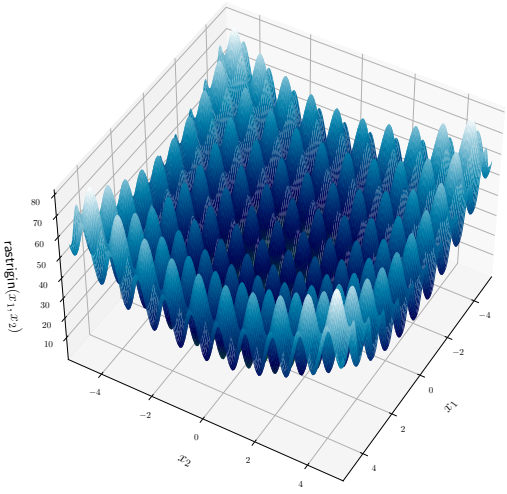


FIGURE 1.1: Plot of the Rastrigin function in two dimensions. The large number of local optima makes it hard for a classical optimization algorithm to find the global optimum. For this reason, the function is often used for benchmarking optimization algorithms. It is also part of the COCO benchmark, which is used in this thesis.

this can be seen when training a neural network with the standard backpropagation algorithm. The only way to increase the explored optimization paths is to restart the training from a different set of initial weight matrices, which is a costly operation. Black-box optimization does not suffer from this issue because of its inherently exploratory focus.

Nonetheless, black-box optimization methods are limited in their application in large-dimensional problems because of a high time complexity. CMA-ES for example, which is an Evolution Strategy (ES) that will be introduced in Section 1.1.2, has a time complexity of $\mathcal{O}(n^2)$. The large number of parameters in deep neural networks thus makes it impossible to use CMA-ES for deep neuroevolution, i.e. the optimization of a large number of neural network parameters. However, we can use black-box optimization for parameter learning. In this scenario, we do not optimize the weights of the neural network but hyperparameters such as the learning rate or weight regularization coefficients.

While the time complexity is a disadvantage, there are also many advantages of black-box optimization techniques. The main advantage over classical optimization is that there exists only one requirement: we should be able to generate a fitness score for each individual. This fitness score can be any arbitrary function, and it does not need to be differentiable. This means that black-box optimization is applicable to any problem as long as we can construct such a fitness function. Although it was stated earlier that black-box optimization is not feasible for deep learning, we can still apply it to smaller networks. Another important use case is Direct Policy Search in Reinforcement Learning (RL). Nowadays, RL problems are usually tackled with the Reinforcement Learning Framework, where the model tries to learn the reward of the agent being in a certain state and doing a particular action. Instead of learning the rewards and using them to find good policies, black-box optimization methods can learn the policy functions directly.

1.1.1 Evolution Strategies

Evolution Strategies are a family of black-box optimization algorithms for solving continuous problems as an alternative to classical optimization methods such as Gradient Descent. Similar to Artificial Neural Networks (ANNs), they are also inspired by nature. If we consider natural evolution as an optimization procedure that has been working for million of years, it is fair to conclude that we should at least try to apply it to our modern day problems. Moreover, it is safe to assume that these ancient algorithms have been improving over time, as Rechenberg (1989) states:

"*Evolution has optimized biological processes. Evolution in itself is a biological process. Ergo: Evolution must have optimized itself. The evolution of evolution is a fact that is not only important for the momentary contribution of an individual to the survival of the species. In the course of several generations better genetic strategies are being selected and improved as well, better in the sense that they promote a faster environmental accomodation.*"

When talking about ESs, it is important to keep in mind that they are multi-agent algorithms, which means that they maintain a set of candidate solutions that is often referred to as a population of individuals. In the context of black-box optimization, an individual consists of multiple parameters corresponding to the dimensionality of the problem. What exactly these parameters are depends on the problem. In Neuroevolution for example, these parameters could be the weights of a neural network, some values defining the network architecture or some other parameters such as which activation function to use. The individual parameters of an individual are often called *genes*, and the specific constitution of these parameters is often referred to as the *genotype*. Once there is a population, each individual is assigned a fitness score that represents how well it did in the context of the task or problem. Such fitness scores can be simple, e.g. the mean absolute error (MAE), or complex and non-differentiable. After computing these fitness scores, a new population is generated based on the performance of the individuals. One of these iterations is called a *generation*. The population is maintained implicitly by retaining the parameters of a distribution from which the population is then sampled in each generation. Then, the parameters of the search distribution are adapted according to the fitness scores.

ESs use a global weighted recombination of the individuals in the population to build a collective direction of improvement. This means that we aim at finding even better individuals in the neighborhood of the last sampled population. The larger the population size λ, the easier it is to find such individuals with a better fitness score. Libraries such as `pycma` (Hansen, Akimoto, and Baudis, 2019) include heuristics for choosing an initial population size so the user does not necessarily need to specify it. In the mentioned example, the population size is chosen to be $\lambda = 4 + 3 \log d$, where d is the dimensionality of the problem. However, Hansen and Kern (2004) mention that tuning this default population size, i.e. increasing it, remarkably improves the performance on 6 out of 8 test functions that they used. In their experiment, the authors repeatedly increased the population size in the sequence 5, 7, 10, 14, 20, 32, 50, $\lfloor 50\sqrt{2} \rfloor$, 100, $\lfloor 100\sqrt{2} \rfloor$, 200, ..., and they conducted 100 runs for each experimental setting.

1.1.2 Covariance Matrix Adaption Evolution Strategy (CMA-ES)

CMA-ES, which stands for Covariance Matrix Adaption Evolution Strategy (Hansen, 1996), is a well-known ES that has been the industry standard for decades. By adapting the covariance matrix, the algorithm learns second-order information about the

objective function, which is similar to the approximation of the inverse Hessian matrix of an objective function in the quasi-Newton method that belongs to classical optimization. Since CMA-ES is a black-box optimization algorithm, it does not make as much assumptions about the fitness function as the quasi-Newton method.

Population sampling. In principle, CMA-ES follows the same strategy as many ESs. Given an initial distribution and its parameters, a population is sampled and evaluated. Then, the fitness scores of the individuals are used to update the distribution parameters, which moves the distribution within the search space. As described in the CMA-ES tutorial (Hansen, 2005), the sampling of the population for generation $g+1$ from the multivariate normal distribution can be described as follows:

$$\boldsymbol{x}_k^{(g+1)} \sim \boldsymbol{m}^{(g)} + \sigma^{(g)} \mathcal{N}\left(\mathbf{0}, \boldsymbol{C}^{(g)}\right) \qquad \text{for } k = 1, \ldots, \lambda \tag{1.2}$$

where $\boldsymbol{x}_k^{(g+1)}$ is the k-th individual of generation $g+1$, $\boldsymbol{m}^{(g)}$ is the mean of the search distribution at generation g, $\sigma^{(g)}$ is the standard deviation or step-size at generation g, $\mathbf{0}$ is the zero vector, $\boldsymbol{C}^{(g)}$ is the covariance matrix at generation g, and λ is the population size.

Updating the mean. To go from generation g to generation $g+1$, we need to compute the mean $\boldsymbol{m}^{(g)}$ and covariance matrix $\boldsymbol{C}^{(g)}$. The update of the mean $\mathbf{m}$ from generation g to generation $g+1$ is given by

$$\boldsymbol{m}^{(g+1)} = \boldsymbol{m}^{(g)} + c_m \sum_{i=1}^{\mu} w_i \left(\boldsymbol{x}_{i:\lambda}^{(g+1)} - \boldsymbol{m}^{(g)}\right) \tag{1.3}$$

where $c_m \leq 1$ is a learning rate, λ and $\mu \leq \lambda$ are the population and parent population size respectively, w_i a weight for the weighted average over μ individuals, and $\boldsymbol{x}_{i:\lambda}^{(g+1)}$ is the i-th best individual w.r.t. the ranking of the fitness scores. For a minimization task, this means that $\boldsymbol{x}_{1:\lambda}^{(g)} \leq \cdots \leq \boldsymbol{x}_{\lambda:\lambda}^{(g)}$ for any generation g. The tutorial mentions that the default learning rate is $c_m = 1$. Since the weights w_i of the weighted average sum up to 1, Equation 1.3 can be simplified if the default learning rate is chosen:

$$\boldsymbol{m}^{(g+1)} = \sum_{i=1}^{\mu} w_i \boldsymbol{x}_{i:\lambda}^{(g+1)} \tag{1.4}$$

The only computation that is needed to do this update of the mean is the evaluation and the ranking of the fitness scores.

Updating the covariance matrix. What takes most computational effort is the update of the covariance matrix $\boldsymbol{C}$. Without going into detail, it can be said that the time complexity for one update of $\boldsymbol{C} \in \mathbb{R}^{d \times d}$ is $\omega(d^2)$ because CMA-ES needs to compute the square root of the covariance matrix. The complexity can be improved to $\mathcal{O}(d^2)$ by using the inverse of the triangular Cholesky factor of $\boldsymbol{C}$ instead, as Krause, Arbonès, and Igel (2016) proposed. Still, the time complexity of CMA-ES is the main drawback of the algorithm and the reason why it cannot be applied large optimization problems as they are common in deep learning.

IPOP. There are many heuristics that can be applied to stop the optimization procedure of CMA-ES early when there is no clear direction of progress. In such a case, the

optimization is usually restarted from another initial search distribution. At the time of the restart, the population size can be changed. This is what Auger and Hansen (2005) do in their IPOP restart strategy, where the population size is increased by a given factor every restart. While the performance compared to a local restart strategy with a constant small population size was similar on unimodal problems, it was increased significantly for 29 out of 60 multimodal problems. There were, however, also 4 multimodal problems where the performance was significantly worse. In most cases, though, it still seems to perform better. Because the most recent data about the performance of CMA-ES on the COCO BBOB suite, which will be introduced in Section 1.2.1, was generated using the IPOP restart strategy with an increment factor of two, all of the experiments in this thesis use this strategy as well.

1.1.3 Hessian Evolution Strategy (HE-ES)

The Hessian Estimation ES (HE-ES) is a Evolution Strategy recently invented by Glasmachers and Krause (2020). In principle, the algorithm is aimed at twice continuously differentiable problems. However, the authors show that the algorithm also works if this assumption is violated. The idea of the algorithm is to directly estimate the inverse of the Hessian matrix of a problem. The reason behind this is that if an ES learns to adapt the covariance matrix to be exactly the inverse of the Hessian, then this is equivalent to learning a transformation that maps the input space to the sphere functions for all convex quadratic problems. Since this sphere function is easy to optimize, this is a desirable outcome. The authors mention that this advantage also extends to local optima in twice continuously differentiable problems due to Taylor's theorem.

The strategy of HE-ES essentially is to estimate the curvature of the fitness function using mirrored samples $m \pm v$ on a line through the current mean m of the search distribution, and by evaluating $f(m + \alpha v)$ for $\alpha \in \{-1, 0, 1\}$. The authors also adapted the sampling of the offspring to one that serves better for the estimation of the curvature of the fitness function. For the estimation of the curvature, an estimation of the second directional derivative of the fitness function in the point m is computed. This estimate converges towards the actual directional derivative of a convex quadratic function, even for a general twice differentiable function, if $\|v\| \to 0$. This quantity can be seen as a component of the Hessian matrix H of the fitness function in direction $\frac{v}{\|v\|}$. By choosing v to be a multiple of the i-th standard basis vector in $\mathbb{R}^d$, this component simplifies to the diagonal entry H_{ii} of the Hessian matrix.

Because one pair of mirrored samples only gives information about the curvature in one single direction, the authors propose a method to sample the next pair of mirrored samples such that it is orthogonal to the previous one. Then, instead of adapting the covariance matrix C such that it matches the curvature in the sampled directions, the authors adapt C to be the closest multiple of the inverse Hessian matrix H^{-1}, which is an optimal covariance matrix as well. To do so, the authors only change the curvature of the covariance matrix to represent the *relative* curvature values that were measured. The authors also ensure that the determinant of the covariance matrix is kept constant, because it encodes the global scale.

To derive the entire HE-ES algorithm is out of the scope of this thesis. It has been found to work well on a broad range of problems despite the rather strong assumptions that were made. The authors state that it works especially well on smooth

unimodal problems, but it can also work well on non-convex and even discontinuous problem. Therefore, HE-ES is a competitive ES and an interesting alternative to CMA-ES.

1.1.4 Block Diagonal Natural Evolution Strategy (BD-NES)

Natural Evolution Strategies (NESs) are a family of evolutionary algorithms that follows the natural gradient of the expected fitness. The fundamental representation of this family is xNES (Glasmachers, Schaul, et al., 2010), with updates based on the natural gradient that are closely related to the various update rules in CMA-ES. Because NES generally have a computational cost of $\mathcal{O}(d^3)$ with d being the dimensionality of a single individual, Schaul, Glasmachers, and Schmidhuber (2011) proposed a Separable NES (SNES), which assumes the parameters to be independent and thus restricts the covariance matrix to be diagonal. Notably, the same principle was also applied to CMA-ES (Ros and Hansen, 2008). This modification reduces the computational cost to $\mathcal{O}(d)$. This change can greatly improve the wall-clock time of the algorithm, i.e. the real time that it takes for the algorithm to converge. However, it also reduces the sample efficiency of the optimization algorithm, i.e. how many individuals or evaluations that it takes to converge. Thus, there is a trade-off between the two approaches. If the separable strategy is too sample inefficient on a specific problem, then the wall-clock time may be the same or even worse than the one for the full covariance strategy. On the other hand, we can have a large sample efficiency but the wall-clock time may be so large that the optimization becomes infeasible.

For this reason, Cuccu (2018) introduced the Block Diagonal NES (BD-NES). The idea is to use a block-diagonal covariance matrix rather than discarding all of the correlations or using the entire correlational information. By specifying the number of blocks m and their size b_i, this approach generalizes both xNES and SNES. For xNES, one can simply choose $m = 1$ and $b_1 = d$, where d is the dimensionality of the problem. For SNES, we set $m = d$ and $b_i = 1$ for all $i \in \{1, \ldots, d\}$. While the computational cost of each block in BD-NES is cubic, it can still be faster than xNES. The reason is that the complexity is now $\mathcal{O}(mb^3)$, where $b = \max_i b_i^3$. Since the number of blocks m is a constant (for any run), the cost is given by $\mathcal{O}(b^3)$, which only depends on the largest block size and not on the entire dimensionality. Another advantage of BD-NES that will be exploited here is that it can be parallelized since different blocks of parameters are assumed to be independent (Cuccu, 2018).

Therefore, the only thing that is needed to optimize the blocks in parallel is a mechanism that evaluates the fitness scores of the population within one block of parameters on the entire problem with all parameters. This can be achieved by using a simple synchronization method that fetches a copy of the current mean of the global search distribution and inserts the individual in the corresponding block. This entire individual is then evaluated by the fitness function. After repeating this procedure for each of the individuals in the population, there is enough data to perform the update of the local search distribution.

To evaluate BD-NES, the author combined two classical RL benchmark tasks into a more complex task: the octopus-arm acrobot task. To solve this task, a network with 3680 weights was designed, which was unfeasible at the time with xNES on hardware typically available at the time. Both BD-NES and SNES were run on the octopus-arm acrobot task and the result was that BD-NES converged with 30% less evaluations than SNES. Moreover, BD-NES also reached comparable results to SNES

in 15% less wall-clock time, without making use of parallel resources nor special hardware.

1.2 COCO: Comparing Continuous Optimizers

COCO stands for Comparing Continuous Optimizers, and it is a benchmark that can be used to assess the performance of various optimization algorithms (Hansen, Brockhoff, et al., 2019). It was developed by Nikolaus Hansen (one of the authors of CMA-ES) and others, mostly within the Numerical Black-Box Optimization (NumBBO) project[1]. Although COCO was originally written in C, it also includes interfaces for Java, Matlab and Python. In the rest of this work, I will only talk about the Python interface, which comes with two modules: `cocoex` and `cocopp`.

`cocoex` contains the benchmark problems and everything else that is needed to measure the performance of the algorithms. `cocopp` is used for post-processing. It automatically generates LaTeX tables and plots that can be used in publications or any other document.

COCO comes with different suites that correspond to classes of problems and it has been used for the Black-Box Optimization Benchmarking (BBOB) workshops at the Genetic and Evolutionary Computation Conference (GECCO) since 2009. The corresponding suite `bbob` is the most commonly used. Each suite contains different functions, which are divided into subgroups of functions having specific properties. The module `cocopp` takes care of generating both the graphs for the different function groups as well as the graphs for the entire set of problems.

1.2.1 BBOB

The BBOB suite comes with 24 noise-free real-parameter single-objective benchmark functions in dimensions 2, 3, 5, 10, 20, and 40 (Hansen, Finck, et al., 2009). All of the functions have their global optimum in $[-5, 5]^d$ by design, where d is the dimensionality of the function. The functions are divided into five subgroups: separable functions (f1-f5), moderate functions (f6-f9), ill-conditioned functions (f10-f14), multi-modal functions (f15-f19), and weakly structured multi-modal functions (f20-f24). For DiBB, which will be introduced in Section 2.1, it is especially important to analyze the difference between the plots of the separable functions, which should show a good performance since the problems are separable, and the plots of the other function groups.

1.2.2 BBOB-Largescale

The BBOB-Largescale suite contains the same 24 same functions that are used in the BBOB suite in dimensions 20, 40, 80, 160, 320, and 640. However, there are some modifications that were made to decrease the cost of a function call in a large-scale setting. More specifically, Elhara et al. (2019) replace the full orthogonal matrices of the rotational transformations with orthogonal transformations that have linear complexity in the problem dimension. Besides that, they normalize the Schaffer (f17, f18), Weierstrass (f16), Gallagher (f21, f22), and Katsuura (f23) functions by the parameter $\lambda(d) = \min(1, 40/d)$, where d is the dimensionality of the function. They do this to have uniform target values that are comparable over a wide range of dimensions. Moreover, the authors state that they generalized the Discus, Bent

[1] For more information, see https://numbbo.gforge.inria.fr/doku.php

Cigar, and the Sharp Ridge functions such that they have a constant proportion of distinct axes that remain consistent with the BBOB suite.

1.2.3 Past Results

The module `cocopp` can be used to access data from other authors on their algorithms. This means that users can compare the performance of their optimizers with previous results without having to rerun the experiments of other authors. However, the amount of data that is available depends on the suite that is being used. For example, there is a lot of data available for the `bbob` suite, but the only author that contributed results to the `bbob-largescale` data archive is Varelas (2019). The `cocopp` module makes it easy to search for previous data and to include it in the generated graphs. However, to get a better overview, I recommend to take a look at the benchmarking data archive website[2], which not only includes the data of other authors but most of the times also related material such as articles, source code and remarks for the specific algorithm.

1.2.4 General Usage

There are three important classes that are used during the benchmarking process, which are included in the `cocoex` module: `Suite`, `Problem`, and `Observer`. Each suite comes with a set of problems. For each problem, we can monitor the number of evaluations and their results using an `Observer`, which saves relevant statistics into files. These can then be used by the `cocopp` module to generate the plots and tables for comparing the performance of different algorithms.

To have a fair comparison, all of the algorithms are usually restricted to a certain number of evaluations per problem. Generally, a budget multiplier b is defined and then multiplied with the dimension d of the problem to get the total number of evaluations m per problem. On their GitHub repository[3] and in their paper (Hansen, Auger, Ros, et al., 2021), the authors of COCO provide sample experiments in various languages that can be used to gain a better intuition on how the benchmark works.

1.2.5 Inner Workings

Each suite does not only contain the problems that were mentioned earlier, but also variations of them. These variations are called *instances* and if we were to count each of these instances as a separate problem, the bbob suite would consist of 360 problems for example. Examples of the differences between the original problem and its instances are a rotated, or translated, search space. These instances are introduced to make the experiments more robust and fairer, for example by preventing an optimizer from exploiting artificial function properties.

COCO measures the performance of an optimizer in terms of runtime. The runtime of an optimizer is defined as the number of function evaluations that have been conducted until a prescribed target value has been hit. If the algorithm does not reach the target value, the runtime is thus undefined. The higher we choose the budget multiplier, the greater the fraction of runs with a defined runtime. The authors of COCO state that while independent restarts from randomized initial solutions are a

[2]The website of the archive is available at https://numbbo.github.io/data-archive/

[3]The examples are available at https://github.com/numbbo/coco/tree/master/code-experiments/build/python

good idea and generally lead to a wider range of solved problems, it does not change the performance assessment in COCO because these restarts also come at a higher runtime.

In contrast to actual restarts, COCO also makes use of *simulated restarts*. To do so, the authors assume that the different instances of a problem are merely random repetitions, i.e. repetitions that could for example be explained by restarting from another randomized initial solution.

Under this premise, the simulated restarts are defined as follows: every time an unsolved problem is encountered, a run from an instance of the same problem is drawn uniformly at random with replacement until we find an instance for which the problem was solved. The sum of the evaluations of the initial unsolved problem and all drawn problem instances is then calculated and assigned to the runtime of the previously unsolved problem. Simulated restarts are either always successful or always unsuccessful. The latter happens if none of the instances of a specific problem had been solved.

1.2.6 Data Format

Every time a problem is evaluated with a given input vector, the `Observer` generates three raw data files with file extensions `.dat`, `.tdat`, and `.rdat`, as well as a summary file with the file extension `.info` that serves as an index for the raw data files. The difference between the `.tdat` and `.dat` files is that the `.dat` files are target-based, while the `.tdat` files are runtime-based. The target-based data files are important for the generation of the plots with `cocopp` and they only keep track of evaluations that improve the best fitness score seen so far. More precisely, they only contain evaluations for which the difference between the optimum function value of that specific problem and the noise-free function value is smaller than the quantity $10^{i/5}$ for any $i \in \mathbb{Z}$.

Once such a target is reached for a specific integer i, none of the evaluations with a noise-free fitness value larger or equal to $10^{i/5}$ is logged anymore. Thus, to generate a new line in the `.dat` file, an optimization algorithm now has to find a candidate solution x that satisfies $x < 10^{(i-1)/5}$.

On the other hand there are the runtime-based `.tdat` files that focus on the runtime, i.e. the number of evaluations, rather than an increasing fitness score. A new line is written to the runtime-based data files for every evaluation number n with $n \in \{\lfloor 10^{i/20} \rfloor \mid i \in \mathbb{Z}\}$. Moreover, the recent evaluation and the best values that have been seen so far are also written into the `.tdat` file once a termination criterion is fulfilled, as Hansen, Auger, Finck, et al. (2010) describe in their experimental setup.

I was able to verify that the runtime-based files are not needed in order to generate the plots with `cocopp`. If there are no such files present, `cocopp` will not work, but if there are empty `.tdat` files present, the graphs are generated anyway. This is also in line with a GitHub issue where it is stated that the `.tdat` files are only used for extracting the maximum budget and for the consistency check of the generated data[4].

While this information may be irrelevant to the user in a local experiment setup, it is important in the case of distributed experiments, since it means that we do not need to merge the `.tdat` files necessarily. In principle, we could just put the last evaluation so that the budget is correctly extracted. However, I still decided to merge the `.tdat` files to make the data more consistent with the original COCO data format.

[4]The issue can be found at https://github.com/numbbo/coco/issues/1445

Depending on the setting, i.e. if a single- or multi-objective suite was used, and if there were additional constraints, the data files should be interpreted slightly different. In this work, I use a suite with single objectives and without any constraints. Moreover, the observations in the experiment were logged in the data format *bbob-new2*. Thus, the following explanation will only cover this specific setting. For any other setting, please refer to the experimental setup mentioned earlier or the documentation on GitHub[5].

Index file. In the context of this thesis, the `.info` file contains information in its header about the suite that was used, a problem identifier, the dimension of the search space, a precision value, an identifier of the algorithm that was used, the name of the logger, an indicator for the data format of the files, and the version of COCO. The second line, which starts with a percent sign, can be used to add an additional comment, e.g. to specify important parameters. Finally, the third and all succeeding lines contain information about the relative location of the raw data files as well as information about the individual runs of the algorithm.

If there are multiple runs, each of them is separated with a comma. The information of each run is of the form `instance:evals|diff`, where `instance` is the instance number of the problem, `evals` the number of function evaluations, and `diff` the difference between the best observed fitness score and the target value for that specific problem.

Target-based and runtime-based data files. The target-based `.dat` and runtime-based `.tdat` data files are structured exactly the same. The data file starts with a header that explains the different columns. The first column contains the function evaluation number, and the second column contains the constraint evaluation number. In the `bbob` and `bbob-largescale` suite that were used in the experiments of this thesis, there were no constraints, thus the second column contained a zero in every row. The third column corresponds to the difference between the best noise-free fitness score that was found so far and the optimum function value plus the sum of all constraint values for that specific individual. Since there were no constraints present in these experiments, this value was just the difference to the target of the function. The optimum function value is also annotated in the header and it does not only depend on the function but also the function instance. In the fourth column we have the measured fitness score of the individual. The fifth column is labeled *best-measured fitness or single-digit g-values*.

Due to the lack of constraints, the fourth and the fifth row were identical in the data generated during the experiments of this thesis. The sixth column and all following contain the individual genes of the individual. Note that this is only true up to dimension 20. After that, the specific genes of the individual are not logged anymore, and only the fitness score is recorded.

Rdat files. The `.rdat` files are labeled as *restart info data files* in the code of COCO[6]. I did not find any documentation on what these files should contain exactly, but in the data generated during the experiments of this thesis, they contained the exact same headers that could be found in in the `.dat` and `.tdat` files and nothing else.

[5]Documentation available at https://github.com/numbbo/coco/tree/master/code-experiments/documentation/data_format.txt

[6]See https://github.com/numbbo/coco/blob/master/code-experiments/src/logger_bbob.c#L69

I believe that they serve to give a better overview over the restarts that were done during the optimization of a specific problem.

For each of the problems in the experiments of this thesis, there was only one line in the corresponding `.rdat` file. In his paper, Hansen (2019) states that he used the IPOP restart scheme for all of the CMA-ES variants that were evaluated. The author also provided the experiment data of his algorithms, which can be easily accessed using `cocopp` or directly downloaded from the benchmark bbob data archive website. I was able to verify that each of the `.rdat` files generated during his experiments also contained only one line. Thus it is valid to have an `.rdat` file with a single line for each problem instance, even if we apply the IPOP restart scheme. This is an important insight for the merging of the distributed experiment data that is described in Section 3.2.2.

1.2.7 Understanding the Plots

An example of an Empirical Cumulative Distribution Function (ECDF) plot that was generated with `cocopp` can be seen in Figure 1.2. The graph includes data of two algorithms and the curve "best 2009", which is an artificial best-case scenario where all of the performances of different algorithms that were run in 2009 are combined and the best performance is used in each data point. Moreover, on the top left we can see that the plot is of the performance on all 24 functions and not only a group of them. Besides the function indices, the dimensionality of the problems is also shown in the top left corner, in this case 40.

The x-axis shows the quantity $\log_{10} \frac{m}{d}$, where m is the number of function evaluations and d is the dimension of the problem. It was mentioned before that the number of total evaluations m that the algorithm could use during optimization was $m = b \cdot d$, where b is the budget multiplier. By substituting $b = \frac{m}{d}$, the quantity on the x-axis can be written as $\log_{10} b$, which is the budget multiplier in log scale. Thus, the ECDF graphs shows how many of the targets were hit given a fixed budget multiplier. The cross on each of the curves denotes the end of the real evaluations and the beginning of the data generated through simulated restarts. From this, we can roughly determine the budget multiplier that was used in the experiments of other authors just by looking at the x-coordinate of the cross. In Figure 1.2 you can see that a budget multiplier larger than 10^5 was used for CMA-ES-20, while the budget multiplier for lq-CMA-ES was somewhere between 10^3 and 10^4.

On the y-axis you can see the fraction of prescribed target values that were reached. Depending on the plot, the value on the y-axis may include multiple functions and the corresponding targets or only a single function and its targets if the plot corresponds to a single function rather than a set of functions. Therefore, the curves of the ECDF plots show how many of the target values were reached, possibly of an entire set of problems, given a specific evaluation budget.

1.3 Distributed Computing

To explain distributed computation, it is easier to start by talking about parallel computation, which usually refers to the execution of code on multiple cores on a local machine (i.e. "in parallel"). The speed of a processor is correlated to the number of transistors that it contains. Since 1975, the number of transistors on microchips doubled every year, which came to be known as Moore's Law. By using more and more transistors, the clock rate of modern processors increased consistently. However, if

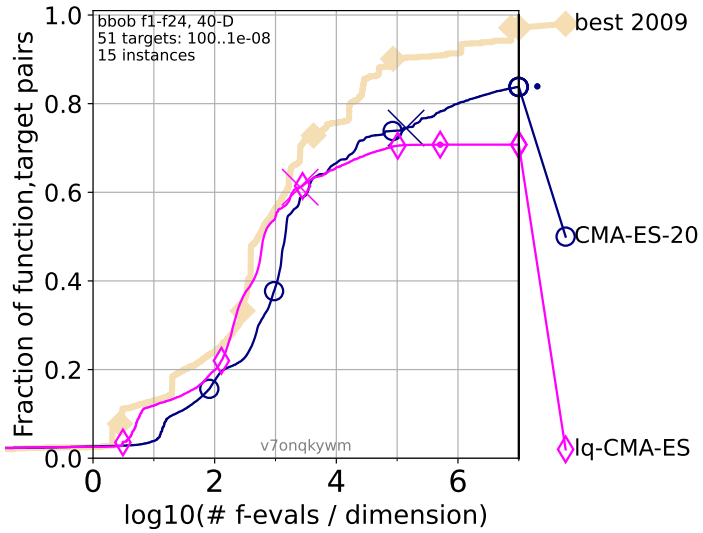


FIGURE 1.2: Example of an ECDF graph generated with cocopp and the data from Hansen (2019). The graph includes data of two algorithms and the curve "best 2009", an artificial best-case scenario where all of the performances of different algorithms that were run in 2009 are combined and the best performance is used in each data point. Moreover, on the top left we can see that the plot is of the performance on all 24 functions and not only a group of them. Besides the function indices, the dimensionality of the problems is also shown in the top left corner, in this case 40.

we stick to the original definition, Moore's Law is bound to expire soon if it has not expired already. The reason behind this is that transistors cannot be made any smaller at some point. Moreover, the computational speed cannot be faster than the speed of light, which shows that there is a physical upper bound to computational performance.

Parallel computing. For this reason, Central Processing Unit (CPU) manufacturers such as Intel and AMD started designing bigger CPUs by combining multiple smaller CPUs. These smaller CPUs are called *cores* and they all run at the same time. By using this novel parallel programming paradigm, computational performance could still keep increasing. However, it also came with a new way of programming that had its own challenges. Programmers had to start thinking about mechanisms to prevent operations from being executed twice, just because two processes may have started addressing the same task almost at the same time.

When talking about processes and CPUs, the topic of threads often comes up. A thread is a context of execution that stores information about how to start a set of instructions. Threads all have access to the same memory within a process and they can be run independently. By scheduling the execution of different threads, a sense of parallelism can be produced even on a machine with a single CPU core. This has to do with the fact that most computations happen so fast that we cannot tell the difference. By performing a set of computations from different processes in sequence, it may still look as if the different processes are running at the same time. Nowadays, processors use both real parallelism by leveraging multiple cores as well as threading.

Distributed computing. It is easy to see how we can increase the computational performance of an algorithm by writing code that can be executed on multiple cores at the same time. However, there is still an upper bound on how much performance we can gain. This upper bound differs from machine to machine and it depends on the number of cores of the processor. To increase the performance further, we could buy a better processor that has more and faster cores. However, there will still be a limit to the number of cores that are included in today's CPUs, and the cost of cutting-edge technology is generally high. Instead, we could simply use more machines. In principle, there is no limit to the number of machines that you can use. Thus, the more parallelizable components an algorithm has, the faster we can make it. This approach of using multiple machines for the execution of a program is called *distributed computing*. A set of computers that is used for distributed computing is called a *cluster*. The individual machines are often referred to as *nodes*, in analogy to the network graph that shows how the machines are connected to each other.

Challenges. While distributed computation has a lot of advantages, it also comes with some challenges. For example, a lot of clusters contain more nodes than necessary, and often data is saved multiple times on different machines. This redundancy ensures that no data is lost when one or more of the machines stop working. Moreover, there has to be some kind of message passing between the nodes of the cluster, unless the computations are all independent from each other.

This has to do with the fact that, in contrast to the CPU cores in parallel computing, there is no shared memory between the machines. Objects, that have to be used by more than one machine, have to be synchronized. In order to perform this synchronization, the objects need to be serializable. This means that the sender should be able to deconstruct the object before sending, and the receiver should know how to reconstruct the object once it was successfully transmitted.

Usually, there are dedicated nodes in the cluster that manage the cluster itself instead of acting like a normal worker node. For example, they can monitor the health of the cluster and check how many nodes are still online. Other dedicated nodes may receive tasks and distribute them to available worker nodes. In the end, it all depends on the architecture and the libraries that are being used, but these principles can be found in almost all settings to some extent.

Data synchronozation. With the advent of big data, many people have started writing distributed algorithms for their applications. This is especially useful for large datasets with a size of several terabytes. Because such a large dataset does not fit into the memory of most machines, they traditionally had to be divided into batches, which where then processed sequentially. In distributed computing, these datasets are also divided into batches, but afterwards they can be processed simultaneously on multiple machines. Indeed, this is such a common use case that there are file systems such as Hbase that have been developed to do exactly that. Another recent trend that many companies have started to adopt is cloud computing where the machines are rented for the time of execution rather than using a local cluster. Some major cloud computing providers are Amazon Web Services (AWS) and Microsoft Azure.

1.3.1 Ray

This work focuses on the study and evaluation of the DiBB framework, as introduced in Section 2.1. Since DiBB internally uses the Ray library, I will briefly discuss

the most important aspects that need to be known to get started with DiBB. Ray is an open source framework for developing distributed applications that is available for Python, Java, and C++ (in an experimental version). It can either be run in a local mode for debugging purposes, or on a cluster of nodes.

Starting Ray. Ray can either be started from the Command-Line Interface (CLI) or directly within the code. But first, the user needs to choose a head node for the cluster. Once chosen, the user executes the command `ray start --head` on the machine. After that, the IP address of the node and a password for the underlying message broker (Redis) is shown. This information can be used to start Ray on the other nodes, thereby connecting them to the head node.

Another and simpler way of setting up a cluster is to make use of YAML configuration files. They contain the information that is necessary for Ray to create a cluster. This information includes the hostnames of the machines that should form part of the cluster, the username and the location of the SSH key that will be used to connect to the machines as well as setup and start commands that should be run on the head and the worker nodes. There are also some more options such as file mounts, which are files or directories that will be copied to all of the nodes in the cluster before starting Ray. An example of such a YAML file that can be used together with DiBB is shown in Listing 2. Once this file is created, the cluster can be managed easily via the Ray CLI. Table 1.1 gives an overview over some of the available commands that are used in conjunction with a YAML configuration file.

Command	Description
`ray attach`	Opens a SSH connection to the head node of the cluster.
`ray submit`	Sends a file to the head node of the cluster and executes it.
`ray dashboard`	Executes the dashboard on the head node and makes it accessible from the local machine through port-forwarding.
`ray up`	Creates or updates a Ray cluster.
`ray down`	Tears a cluster down.
`ray monitor`	Shows the output that would be shown when executing `ray status` on the head node. This command shows the health status of the nodes in the cluster as well as the usage of resources and their demands.

TABLE 1.1: Some of the Ray CLI commands that can be used together with YAML configuration files. They all need to be followed by the path to the YAML file.

Serialization. For the synchronization of objects between nodes, Ray uses its own Plasma object store. This is an in-memory store developed by Apache Arrow. The Plasma store only stores immutable data. Unless it has to be accessed by another node, all of the data on a node remains local. Moreover, Ray uses a customized version of the Pickle 5 protocol to serialize a wide range of objects. It is optimized for the use with NumPy arrays, which makes Ray a perfect tool for many Machine Learning (ML) problems.

One can either use *tasks* (functions) or *actors* (classes) to parallelize the code. In both cases it requires adding the `@ray.remote` decorator above the definition of either the function or the class. If an actor is used by more than one node, it needs

to be serializable. Simple classes that only store built-in data structures are usually serializable by default. But there are also some classes that are not. Examples are the classes `Suite`, `Problem`, and `Observer` from the `cocoex` module that is discussed in Chapter 1. For those classes, there are different ways of writing custom serialization code, as proposed in the documentation of Ray[7]. One of the simplest ways is to add a wrapper class that implements a custom `__reduce__` method.

Calling tasks and actor methods. It is important to note that tasks and actors cannot be instantiated or called the classical way. Instead, the `remote()` method has to be called upon the target. Listing 1 shows how a ray actor can be instantiated and how the class methods are called. Note that on line 11, `ray.get()` is called. This tells Ray to wait for the result of task and then collect it from the object store. Every time a task is executed, it immediately returns a ray `ObjectRef` (an object ID). Besides using `ray.get()` on one or a list of object references, it is also possible to wait for the termination of a task using `ray.wait()`. The advantage `ray.wait()` is that it allows to specify how many tasks, in a list of object references, should be waited upon. A timeout can also be specified after which Ray does not wait for the termination of the tasks anymore but returns. If a result of a task is passed to another task, there is no need to call `ray.get()`, as the object reference can just be treated as if the object itself. Ray will take care of fetching the result once it is available. This is just one of many design principles stated in the documentation.

```
1  import random
2  import ray
3
4  @ray.remote(resources={'SealedBox': 1, 'GeigerCounter': 1})
5  class SchroedingersCat:
6      def is_alive(self):
7          return bool(random.randint(0, 1))
8
9  ray.init(address='auto')
10 cat = SchroedingersCat.remote()
11 print(ray.get(cat.is_alive.remote()))
```

LISTING 1: Example of how to use an actor with Ray to reproduce Schrödinger's cat experiment. On line 4, the resources for the `SchroedingersCat` actor are specified. To create such an actor, there needs to be at least one machine in the cluster that started Ray with a minimum of one `SealedBox` and one `GeigerCounter` resource. Line 9 connects Ray to the already running Ray process on the current machine. Line 10 shows how an actor is initialized using the `remote()` method. In line 11, a class method of the actor is first called using the `remote()` function, which returns an `ObjectRef`. This object id is then passed to `ray.get()` to fetch the actual result of the computation.

1.4 Contributions

My contributions in this thesis are twofold. First, I extended DiBB with some new features that were needed for the experiments in this thesis. Second, I implemented

[7] See https://docs.ray.io/en/master/ray-design-patterns/index.html

a distributed version of the COCO experiments that was suitable for DiBB, including the merging of the experiment data generated by the COCO `Observer` class. I added the following features to DiBB:

- More evenly distributed blocks when there is only a number of blocks specified rather than all of the block sizes
- The possibility to initialize `BlockWorkers` and `FitnessEvaluators` one specified machines in the cluster
- Named actors that make it easier to analyze the Ray dashboard
- The option to choose if DiBB is minimizing or maximizing the fitness function
- Improved sleeping mechanism in `BlockWorkers` that does not block when `reset()` is called
- Automatically choosing the number of `FitnessEvaluators` based on the number of CPU cores if requested
- `Hooks` that are executed within the `BlockWorkers` before starting another generation and within the `FitnessEvaluators` before evaluating the fitness function
- The hook `EvalBudgetLimiter`, which stops the optimization procedure once a given budget of evaluations is reached
- A distributed test case using a 100d Rastrigin function and 20 blocks of size 5

For the integration of DiBB in the COCO experiments, my contributions were as follows:

- Writing a serializable `FitnessWrapper` that includes the ID of the COCO problem and the options for the `Observer`
- A custom hook `FinishOnTargetHit` that stops the optimization of a problem once the final target has been hit, similarly to what is already implemented in the local setting
- IPOP restarts using the `reset()` method of DiBB
- A merging script that aggregates all of the generated data across different `BlockWorkers` (i.e. nodes in the cluster) and runs

Chapter 2

Method

2.1 From BD-NES to Distributed Black-Box Optimization

In Section 1.1.4 we have seen that BD-NES could in principle be parallelized to further improve the reduction in wall-clock time. The idea of separable optimization algorithms is not something unique to the NES family or CMA-ES, but rather something that can be applied to a wide range of optimizers. Similarly, the concept behind BD-NES is not restricted to NESs neither. This means that other optimization algorithms such as CMA-ES could benefit from the advantages of using block-diagonal covariance matrices, or separate groups of presumably correlated variables in general, as well. For this reason, a framework for Distributed Black-Box Optimization (DiBB) was developed by Cuccu in 2019-2021. It allows the user to split problem dimensions into a set of blocks with arbitrary sizes, corresponding to the original dimensionality of the problem.

DiBB also takes care of the two synchronization steps explained earlier, and it comes with wrapper for the CMA-ES algorithm out-of-the-box. For any other optimizer, a simple wrapper needs to implement the methods `stop`, `ask`, and `tell`. The method `stop` should return whether the optimizer has terminated its run. To fetch a new population from the optimizer, DiBB uses the `ask` method. The method `tell` receives a list of individuals and a list of their corresponding fitness scores and performs the update step of the algorithm. Finally, the wrapper should also implement the method `mean`, which returns the current mean of the search distribution.

2.2 Using Ray for DiBB

To distribute the computation of the blocks, DiBB uses Ray. Section 1.3.1 provides a brief introduction into Ray. Now we will look at the specific parts of Ray that are important for the usage of DiBB. Besides the Ray cluster setup, that has to be done by the user, the usage of DiBB should not require a deep understanding of how Ray works. However, there are some scenarios where it is useful to know more about the interaction between DiBB and Ray. For these cases, it might be useful to know which parts of DiBB are Ray actors.

First of all, there are are the the two actors `BlockWorker` and `FitnessEvaluator`. Every method that is called upon one of these instances must be called with the `remote()` method as described in Section 1.3.1. Then, there are the actors `RemoteList` and `RemoteDict`, which are part of the module `dibb.remote_structures`. They are internally used to store the global mean of the search distribution (`RemoteList`) and to pass messages between the actors in one single dictionary (`RemoteDict`). Knowing about this communication dictionary is important to write custom hooks that are executed at specific moments during the optimization. A more detailed introduction

into hooks will be given in Section 2.5.4, but for now it is important to know that the communication dictionary will be injected into each hook. This allows the hooks to communicate with other elements of DiBB.

An example of such a hook that I implemented is the `EvalBudgetLimiter`, which uses the communication dictionary to store the number of evaluations that each `BlockWorkers` has performed so far. Once a total number of evaluations would be reached by starting a new generation on a specific `BlockWorker`, the hook sends a termination signal to the `BlockWorkers`, which will result in the termination of the optimization procedure.

2.3 Cluster Setup

There are multiple ways to set up the Ray cluster that will be used by DiBB. To set the cluster up manually, each of the machines require the same version of Python and Ray installed. Also, DiBB and all other dependencies that are used during an experiment should be installed first. The Ray process can either be started from within the code or using the CLI. The latter option is often the preferable one, because it decouples the existence of the cluster from the execution of the code.

When starting Ray on the worker machines, it is important to add the resource `DedicatedMachine` using the the option `--resources`. The idea is that each machine contains at most one `BlockWorker`. Adding more than one `DedicatedMachine` resources is not recommended. When using the `FitnessEvaluators`, DiBB can automatically spawn as many of them as there are cores on the machine that hosts the `BlockWorkers` if the user sets the parameter `nfiteval` to *auto*. If the machine hosts more than one `BlockWorker`, this might lead to issues. A typical command to start Ray on the head node is the following:

```
ulimit -c unlimited && ray start --head --port=6379
```

Note that the `ulimit -c unlimited` removes the maximum number of opened files. This is needed because Ray opens many connections between worker processes, which can quickly consume a large number of file descriptors[1]. Running the command on the head node should automatically print the command that can be used to connect the worker nodes to the head node. The final command will be similar to the following one:

```
ray start --address='172.217.168.46:6379'
 ↪  --redis-password='5241590000000000'
 ↪  --resources='{"DedicatedMachine": 1.0}'
```

Instead of doing this manually, Ray provides once again the option to use configuration files in a YAML format, which are then used in conjunction with CLI commands. Listing 2 shows a minimal example of such a YAML file that can be used to start a simple Ray cluster that can be used by DiBB. For information about additional options and how to use the Ray cluster with AWS, Azure or GCP, please refer to the documentation of Ray[2].

[1] See https://docs.ray.io/en/master/cluster/guide.html?highlight=ulimit#system-configuration

[2] The ray cluster section of their documentation is available at https://docs.ray.io/en/master/cluster/index.html

```yaml
cluster_name: dibb_cluster_example

provider:
    type: local
    head_ip: hostname_01
    worker_ips:
    - hostname_02
    - hostname_03

auth:
    ssh_user: ray_cluster_user
    ssh_private_key: ~/.ssh/id_rsa

head_start_ray_commands:
    - ray stop
    - ulimit -c unlimited && ray start --head --port=6379
     ↪ --autoscaling-config=~/ray_bootstrap_config.yaml

worker_start_ray_commands:
    - ray stop
    - ray start --address=$RAY_HEAD_IP:6379
     ↪ --resources='{"DedicatedMachine":1}'
```

LISTING 2: Example of a cluster configuration file for Ray. The hostnames of the form `hostname_01` need to be replaced either by the hostnames as specified in the ssh config file, or the IP addresses of the machine. The `ray_cluster_user` also needs to be replaced with the user that will be connecting to the machines via ssh. Depending on how the authentication in the cluster is set up, the path to the ssh private key may also change.

2.4 Inner Structure of DiBB

Once DiBB is initialized, it creates a number of `BlockWorkers` (BWs). Each of these `BlockWorkers` is initialized in principle on a separate machine. Both DiBB and all of the `BlockWorkers` have access to the shared communication dictionary, which is a Ray actor. This dictionary stores the state of every BW, i.e. if they are running or paused, a termination flag, the best individual seen so far and the best fitness score associated to the best individual.

Depending on the initialization parameters, DiBB will be running with or without `FitnessEvaluators` (FEs). There are two advantages of having FEs. First, they run in parallel on the available cores. This is especially useful if the time it takes to evaluate the fitness function is large. Second, they are isolated from each other besides the Ray communication mechanisms. An example case, in which this might be useful is when the fitness function needs to construct its own environment for a RL task.

The drawback of using `FitnessEvaluators` is that there is a certain overhead introduced by using Ray actors instead of normal function calls. For this reason, it is

recommended to use DiBB without `FitnessEvaluators` whenever the fitness function can be evaluated fast and if there is no need to have a set of isolated fitness functions. Moreover, this is also the reason why DiBB does not use `FitnessEvaluators` by default.

After the user calls the `run` method on the DiBB instance with a certain number of generations, DiBB forwards the `run` command to all of the BWs. Let us take a look at the actions performed by the i-th `BlockWorker` in each generation. Two search distributions are important for this BW: the global one with a d-dimensional mean, and the local one with b_i-dimensional mean and a covariance matrix $\mathbf{C} \in \mathbb{R}^{b_i \times b_i}$. Here, d stands for the dimensionality of the problem and b_i stands for the dimensionality of the i-th block. Covariance matrices are only computed for the blocks but not for the entire problem, since we want to restrict the entire covariance matrix to be block-diagonal. The local mean of the search distribution corresponds to the i-th block of parameters in the global mean. At the beginning of a generation, a copy of the global mean is fetched and stored on the BW. I will refer to this copy as *full individual* in the following. Each of the individuals in the current population is inserted into the i-th block in the full individual. This modified full individual is then evaluated with the fitness function. The fitness scores are then passed to the optimizer to update the parameters of the local search distribution. Afterward, the global mean (not the copy) is updated by replacing the i-th block with the current mean of the local search distribution after the updates.

Figure 2.1 illustrates this process for the setting with FEs. Whenever the optimizer of a `BlockWorker` reports a stop, the `BlockWorker` is paused for five seconds. The idea is that there are other `BlockWorkers` that might have performed an update of the global mean during these five seconds. This modification of the global mean also leads to other fitness scores for the previously evaluated population, which means that the stopping criterion might not hold anymore. DiBB periodically checks the state of all of the `BlockWorkers`. Once all of them are paused, this means that all of the optimizers converged. A termination signal is then sent to all BWs, and the optimization procedure comes to an end.

2.5 DiBB Usage

In this section, I show how DiBB can be used in practice. There are many parameters of DiBB that are optional. However, in some cases, they are needed to change the behavior of DiBB, which is why I briefly explain all of them. I also explain, the `reset()` method, how to use custom `Hooks`, and I show a minimal working example of an experiment with DiBB.

2.5.1 Initializing DiBB

DiBB, can be either initialized with a list of different block sizes or a number of blocks. In the latter case, the block sizes are chosen to be uniform. In the next step, DiBB initializes a number of `BlockWorkers`, which in turn may initialize a certain number of `FitnessEvaluators`. The `FitnessEvaluators` are only used if the parameter `nfiteval` is specified by the user. Otherwise, the evaluations of the fitness function will be performed on the `BlockWorker`sequentially. Each `BlockWorker`uses one Ray resource of type `DedicatedMachine`, i.e. there is at most one `BlockWorker`running on any particular node in the cluster. To do so, the worker nodes must have been started with one `DedicatedMachine` resource each. The `FitnessEvaluators` on the

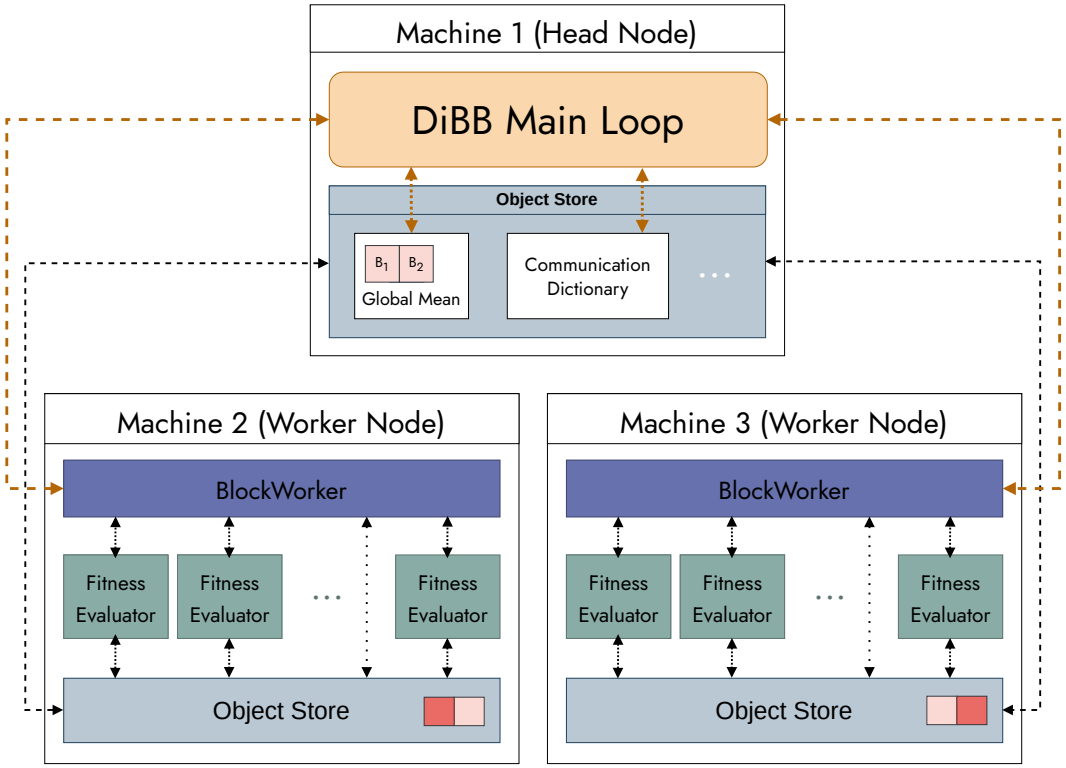


FIGURE 2.1: Illustration of the interaction between local search distributions in each `BlockWorker` and the global mean of the search distribution stored on the head node. In this example, only two `BlockWorkers` are used. In each generation, the i-th `BlockWorker` samples another population and fetches a local copy of the global mean (full individual). For each of the individuals in the population, the i-th block of the full individual is replaced with the corresponding individual, and the fitness score of this modified full individual is evaluated. These scores are then used to update the parameters of the local search distribution, and the updated mean of the local search distribution is sent back and stored in the i-th block of the global mean.

other hand, if they are being used, are all spawned on the same machine as the corresponding `BlockWorker`. This makes sure that the communication overhead that is introduced by using multiple `FitnessEvaluators` is as small as possible.

Before DiBB can be initialized, there are some other parameters that have to be specified. The first one is the fitness function that will be used in the black-box optimization procedure. It can either be a function or a string. If a string is provided, it will be passed to the built-in `eval()` function of Python, which parses the string and evaluates it as a Python expression. When DiBB receives such a string as the fitness function, it requires the evaluated expression to return an actual function. The advantage of this approach is that additional code can be injected into the `FitnessEvaluators` or `BlockWorkers`.

The second parameter is a string denoting the optimizer that should be used. For now, CMA-ES (`cma-es`) is supported out-of-the-box. To use another optimizer, a wrapper can be written and integrated into DiBB. The important part is that the algorithm supports an ask-tell interface: The `BlockWorkers` will ask the optimizer to return a population of individuals from the current search space, which are then evaluated either on the `BlockWorkers` or the `FitnessEvaluators`. Afterward, the `BlockWorkers` tell the optimizer the individual fitness scores, which are used to update the parameters of the search distribution. In many modern ESs, this interface is already included. If this is the case, there are only minor changes that need to be done when writing the wrapper class for a given optimizer. A more detailed explanation of the required and the optional parameters can be found in Table 2.1.

Another argument that may be specified is a dictionary of hooks. For now, the dictionary may only use the `BlockWorker` and `FitnessEvaluator` keys. Each value in the dictionary can be set to a list of hooks. These hooks should inherit from the `Hook` class that is included in DiBB, and implement the `execute_on` method. Each of these hooks is executed before the main functionality of the corresponding class. For the `FitnessEvaluators`, the hooks are run just before executing the evaluation of the fitness function. The `BlockWorkers` execute the hooks just before running another generation.

Each hook passes first through DiBB before being stored on the respective class instance. DiBB injects the communication dictionary that it uses to communicate with the `BlockWorkers`. The hooks may then use this dictionary to communicate with the DiBB instance, the `BlockWorkers`, and even with other hook instances. DiBB already includes the `EvalBudgetLimiter`, which is a hook related to the `BlockWorkers`. It can be used to set a fixed budget of evaluations for the fitness function. The `BlockWorkers` then check if a a new generation is still feasible given the evaluation budget and the population size. If the budget would be exceeded, the generation is not executed anymore and the `BlockWorkers` stop the optimization procedure. Another advantage of the `EvalBudgetLimiter` is that it allows us to check the current number of evaluations per block.

2.5.2 Running DiBB

Calling the `run()` method with a number of generations will start the optimization procedure. The optimization finishes once all of the `BlockWorkers` performed the specified number of generations or if a stopping condition of the optimizer is met. Note that each block may fulfill another stopping criterion. Only if all blocks are paused due to a stopping condition, the optimization finishes early.

An example of how to initialize and run DiBB on the Schaffer function can be seen in Listing 3. After the optimization procedure has ended, the result can be

Parameter	Default	Description
`init_ind`	✗	Flattened array containing the initial candidate solution.
`blocks`	✗	Either an integer corresponding to the number of blocks or a list of integers corresponding to the size of each block. If only a number of blocks is specified, the dimension of the initial individual will be split and distributed uniformly.
`optimizer`	✗	String identifier of the wrapper class for a specific optimizer. Use `cma-es` to select the CMA-ES algorithm. Custom wrappers are also supported, see the *readme* file in `dibb/opt-wrappers`.
`fit_fn`	✗	The fitness function that will be used during optimization. Can either be a function or a string that evaluates to a function when passed to Python's built-in `eval` function.
`optimizer_options`	`{}`	Dictionary containing the options for the optimizer. These depend on the optimizer that is being used. For CMA-ES, these options may include the initial sigma as well as the options of the `pycma` module.
`available_machines`	`None`	A list of IP addresses of worker machines in the cluster. If specified, the `BlockWorkers` will be initialized on these machines. Therefore, the number of specified machines must match the number of blocks.
`minimization`	`True`	Boolean that specifies if the fitness function should be minimized. If `False`, larger fitness scores are considered better.
`nfiteval`	`None`	Number of `FitnessEvaluator` that are created for each block of parameters. If `nfiteval` is `None`, the fitness score of the individuals will be evaluated sequentially on the `BlockWorker`. If `nfiteval` is *auto*, the number of `FitnessEvaluators` that will be used is computed automatically depending on the number of cores of the machine.
`ntrials_per_ind`	`1`	Number of fitness evaluations that are performed for each individual. Useful for non-deterministic fitness functions.
`fit_aggregator`	`np.mean`	Function that is used to aggregate the results of multiple trials per individual. Only important if `ntrials_per_ind` > 1.
`verbose`	`True`	Boolean specifying whether additional information should be printed during optimization.
`print_fn`	`None`	A function that accepts one parameter that corresponds to the DiBB instance. This function can then be used to print statistics or other information while waiting for the `BlockWorkers` to finish.
`hooks`	`{}`	A dictionary of hooks that can be applied to the `BlockWorkers` or `FitnessEvaluators`. The keys of the dictionary can be `BlockWorker` and/or `FitnessEvaluator`, and they store a list of hooks that are executed either before running another generation in the case of a `BlockWorker` or before evaluating the fitness function in the case of the `FitnessEvaluator`.
`rseed`	`None`	A seed for the generation of random numbers. Fixing it increases the reproducibility of the experiment.

TABLE 2.1: Required and optional parameters for **initializing** DiBB.

fetched using the method `result()`, which returns a dictionary containing the mean of the search distribution (`mean`), the best individual seen during optimization (`best_ind`), and the corresponding fitness score (`best_fit`).

```
1  import ray
2  import cma
3  import numpy as np
4  from dibb import DiBB
5
6  ray.init(address='auto')
7
8  dibb = DiBB(init_ind=np.random.uniform(-4, 4, 10),
9              blocks=2,
10             optimizer='cma-es',
11             optimizer_options={
12                 'init_sigma': 2,
13             },
14             available_machines=[
15                 '172.217.168.46',
16                 '172.217.168.47'
17             ],
18             fit_fn=cma.ff.schaffer)
19
20 dibb.run(10_000)
21 result = dibb.result()
```

LISTING 3: Example of how to initialize and run DiBB on the Schaffer function, which is part of the BBOB and BBOB-Largescale benchmark. On line 6, a connection to an existing Ray cluster is established. Lines 8 to 18 show the initialization of DiBB. The initial individual is chosen uniformly at random in $[-4, 4]^{10}$. DiBB uses CMA-ES with an initial standard deviation of 2 and two parameter blocks of size 5. Line 20 starts the optimization procedure with a 10 000 generations at most. Afterward, the result of the optimization is fetched in line 21.

2.5.3 Resetting DiBB

DiBB also includes a reset method, which is useful if the optimization task consists of many smaller problems. Without this feature, DiBB had to be reinitialized for every sub-task, resulting in the destruction and creation of various objects in the object stores across a cluster. To avoid this overhead, we can simply reset the fitness function. Other use cases are the dynamical adding or removing of hooks or changing the fitness function to solve a sequence of increasingly harder problems where each problem might benefit from the previously found solution in terms of a warm start with this specific initial individual. The parameters to this method are similar to the ones that have to be specified for the initialization of DiBB. However, leaving some of them out may result in a different behavior, which is why Table 2.2 includes an overview over the parameters of the `reset()` method and their effects on the optimization procedure.

Listing 4 shows how to perform restarts using the `reset()` method. In this example, 10 restarts are performed. In practice, it could be useful to perform the restarts only if a certain condition is met, rather than using a fixed number of restarts.

```
for _ in range(10):
    dibb.reset(init_ind=np.random.uniform(-4, 4, 10),
               optimizer='cma-es')
    dibb.run(10_000)
```

LISTING 4: Example of how to perform restarts using the `reset()` method of DiBB. The code assumes that Ray and DiBB were previously initialized. We see that 10 restarts are performed, and that the initial individual is chosen uniformly at random from $[-4,4]^{10}$ for each restart.

Parameter	Default	Description
`optimizer`	✗	The name of the optimizer (wrapper) to use. For more details, refer to Table 2.1.
`fit_fn`	`None`	Specify a new fitness function. If `None`, the existing fitness function will be used. Whenever a new fitness function is specified, the best individual and the corresponding fitness score from the last run are discarded.
`init_ind`	`None`	Specify a new initial candidate solution. If `None`, the existing mean of the previous search distribution will be used, which should be considered with care when the fitness function has changed.
`optimizer_options`	`{}`	Same as Table 2.1.
`minimization`	`True`	Same as Table 2.1.
`ntrials_per_ind`	`1`	Same as Table 2.1.
`fit_aggregator`	`np.mean`	Same as Table 2.1.
`verbose`	`True`	Same as Table 2.1.
`hooks`	`{}`	Same as Table 2.1.
`rseed`	`None`	Same as Table 2.1.

TABLE 2.2: Required and optional parameters for resetting DiBB.

2.5.4 Writing Custom Hooks

To create a custom hook, a new class has to be written that inherits from `dibb.hooks.Hook`. It needs to implement the `execute_on()` method, which will be executed either on a `BlockWorker` or a `FitnessEvaluator`. The hooks have access to the communication dictionary via `self.comm`. Moreover, the corresponding Ray actor (either `BlockWorker` or `FitnessEvaluator`) is passed as an argument to the `execute_on()` method.

Listing 5 shows how to write and use a custom hook that increases the population size of CMA-ES after a certain number of generations. In this example, the communication dictionary is not used because there is no need to communicate with other `BlockWorkers`. However, it might be important for other use cases.

```
1  import ray
2  import numpy as np
3  from dibb import DiBB
4  from dibb.hooks import Hook
5  import cma
6
7  ray.init(address='auto')
8
9  class PopsizeIncrement(Hook):
10     def __init__(self, increment_factor, ngens):
11         self.increment_factor = increment_factor
12         self.ngens = ngens
13         self.generation_counter = 0
14
15     def execute_on(self, bw):
16         self.generation_counter += 1
17         if self.generation_counter == self.ngens:
18             old_popsize = bw.optimizer.es.popsize
19             new_popsize = int(old_popsize * self.increment_factor)
20             bw.optimizer.es.opts.set({'popsize': new_popsize})
21
22 dibb = DiBB(init_ind=np.random.uniform(-4, 4, 10),
23             blocks=2,
24             optimizer='cma-es',
25             available_machines=[
26                 '172.217.168.46',
27                 '172.217.168.47'
28             ],
29             fit_fn=cma.ff.schaffer,
30             hooks={'BlockWorker': [PopsizeIncrement(2, 300)]})
31
32 dibb.run(10_000)
33 result = dibb.result()
```

LISTING 5: Example on how to write a custom hook that increases the population size of CMA-ES after a certain number of generations. The initialization is almost the same as in Listing 3. The hook is implemented in lines 9 to 20. The `execute_on()` method checks if a specified number of generations is reached. If so, it multiplies the population size of the CMA-ES optimizer on the `BlockWorker` by a given factor. Line 30 tells DiBB that the `BlockWorkers` should use the hook. In this example, the population size on each `BlockWorker` is multiplied by two after the first 300 generations.

Chapter 3

Experiments

3.1 Research Questions

To evaluate the performance and practicality of DiBB, the following three research questions were posed:

1. How does DiBB compare to optimization algorithms that use the entire correlational information?
2. How much faster can DiBB run an experiment with a fixed budget of function evaluations?
3. Does DiBB's performance scale to high-dimensional problems?

The goal of this chapter is to answer these three questions by analyzing the results of four experiments. The chapter is structured as follows: First, I will talk about the experimental setup, which includes the machines and the software that was used, as well as the strategy that was applied to merge the generated data. Then, I will explain the four experiments. Finally, the results of the experiments are reported.

3.2 Experimental Setup

In this section I will talk about the experimental setup that includes the machines and the software in the cluster. Afterward, I present the merging strategy that I implemented to aggregate the experimental data across the different `BlockWorkers` (i.e. worker nodes) and the different runs.

3.2.1 Cluster Setup

Machines. The cluster that was used to conduct the experiments in this thesis consisted of 24 machines that had 8 CPU cores each and 32 GB of RAM. The machines were old and if it had not been for the sake of these experiments, they would have been already disposed of. Therefore, the timing data in the experiments can still be improved by simply using better machines. However, the purpose of the experiments was to answer the research questions, which can be done using the 24 machines that I had access to. All of the machines were running Ubuntu 20.04.2. Additionally, for two of the seven BBOB experiments, I used two better machines that had 32 CPU cores and 128 GB of RAM each. I did this to run as much experiments in parallel as possible.

Software. I had to install the same software on each of the machines in the cluster. All of the Python dependencies were installed inside `pipenv` environments, which is a packaging tool that creates and manages virtual environments similar to `pip` and `virtualenv`, but with some improvements. For each environment, `pipenv` generates two files: a `Pipfile` and a `Pipfile.lock` that keeps track of the dependency tree. First of all, the code for COCO and DiBB had to be transferred to the machines. Building COCO requires Python 2, but afterwards it can be used with Python 3. To build COCO, the following dependencies have to be installed: `build-essential`, `python-dev` and `python-setuptools`. To use `cocopp` for generating the plots and tables from the experiment data, the modules `numpy`, `scipy`, `matplotlib`, and `six` are required.

For each machine, I created a `pipenv` environment directly in the home directory of the user. The environment had to use the Python 3.8.10 interpreter, since Ray only works if the exact same versions of Python and Ray are installed on all nodes in the cluster. The main requirements of the `pipenv` environment were `dibb`, `cocoex`, `cocopp`, `ray` 1.5.1, `pycma` 3.1.0, and `numpy`. Moreover, I used the module `tqdm` to print the progress of the experiments, the `brutelogger` module to save the standard output, and the module `pyinstrument` to profile the running times of DiBB. After each experiment was finished, a notification was sent using the `notify-run` module.

3.2.2 Data Merging

While COCO was designed to be used with any solver, it was not designed to be used in a distributed setting. To use DiBB on COCO, there are some points that need to be considered. The first challenge arises because the three main classes `Suite`, `Problem`, and `Observer` are not serializable by default. For the benchmarking experiments that will be presented in Section 3.3, I wrote a Ray actor `RemCoco` to start and maintain the main execution loop of an experiment in a single dimension. Each `RemCoco` object has its own suite of problems and is only instantiated once on a remote machine, therefore there was no need to make the suite serializable. However, the problem and the observer still needed to be serializable.

For the experiments, I wrote a wrapper class that stored the ID of a problem and the options of the corresponding `Observer`. These options are needed to specify the output directory of the generated data. To work like a normal function, the wrapper class implements the `__call__()` method. The first time that a `FitnessEvaluator` or `BlockWorker` calls this method, the problem and the `Observer` are initialized. I did this to prevent the problem and the `Observer` from being initialized and destroyed more than necessary while being passed from DiBB to the `BlockWorkers`, and possibly also to the `FitnessEvaluators`. Moreover, since the `Observer` is only initialized when it is needed, it prevents the creation of empty directories that are never used.

Each `BlockWorker` is running on a separate machine, together with the corresponding `FitnessEvaluators`. Because every `BlockWorker` or `FitnessEvaluator` maintains its own instance of the `Problem` and `Observer` class from the `cocoex` module, the observed data will be written to disk on the machine that runs the `BlockWorker`. Thus, when using m `BlockWorkers`, the data of the experiments will be stored across m nodes in the cluster. In order to use `cocopp` after the distributed experiments, the data needs to be merged correctly. The merging of the experiment data of this thesis can be divided into two stages: 1) Merging the data of one run on one problem across all `BlockWorkers` and possibly also across their `FitnessEvaluators`, and 2) Merging the multiple runs on one problem into one run.

Directory structure. The observed data of a specific problem in the COCO suites is written to the folder specified in the `result_folder` options that is passed to the `Observer` at its initialization. In the experiments of this thesis, the result folder was chosen to include the following information: ID of the problem (consisting of the name of the COCO suite, the function number, the instance number, and the dimensionality of the problem), hostname of the machine running the `BlockWorker`, block id, and run number. An example of such a generated directory is the following: `bbob_f017_i03_d20_node-02_b00_r01`. In this case, the directory stores the data of the 3rd instance of the 17th problem in the BBOB suite with 20 dimensions on the `BlockWorker` with id 0 and the first run of the problem. Moreover, we know that the problem was evaluated by the `BlockWorker` on machine `node-02`, which makes it easier to check if the data was evaluated correctly on the machines that it should have been evaluated on.

In principle, it is possible to include different dimensionalities in one single suite. However, for the sake of separability of the experiments, I chose to use the suites always with one single dimensionality. Different experiments thus used different instances of the `Suite` class that were initialized with different suite options.

When using DiBB with `FitnessEvaluators` on COCO, the directories have suffixes of the form `-001`, `-002`, etc.: the first `FitnessEvaluator` will create the directory without suffix, the second one will add the suffix `-001`, and so on. It is also important to note that restarts should be done by using the `reset` method of DiBB, otherwise the `Observers` may be reinitialized many times, which leads to large directory suffixes. This is bad because once a problem uses the entire 999 suffixes, COCO will stop working.

After downloading the data from the machines in the cluster to a local directory, the merging process can start.

Merging the data within one run. In the following, I will explain how the `.dat` and `.tdat` files within one run are merged for the setting where there evaluations are performed directly on the `BlockWorker` without any `FitnessEvaluator`. However, it is easy to adapt the merging strategy to include the `FitnessEvaluators` as well, as it corresponds to using one `FitnessEvaluator`. If more than one `FitnessEvaluator` is used, the only difference is that the pool of directories that we currently look at in the merging process is larger.

The first step is to loop through all of the problem IDs. For each problem, we compile a list of directories that contain the data of interest. Then, we group these directories into the different runs that were performed. For each of the runs, we then merge the data. The exact merging strategy is can be seen in Algorithm 1.

Note that the merging of the `.info` and `.rdat` files are not included in the algorithm to save space and because they are trivial, as we can just copy one of the `.rdat` and `.info` files into the output directory. The `.info` file will be corrected during the second merge pass. Because the `BlockWorkers` of DiBB run at the same time, there is no real sequential order. When the best fitness score seen so far was improved multiple times within the same evaluation number across different `BlockWorkers`, I log the improved fitness scores in the order of the magnitude of the improvement. This was done do be as fair as possible.

Merging the data across runs. To merge the data across multiple runs, the `.dat` and `.tdat` files are first concatenated in the order of the runs. The header is only included once at the beginning. Then, the last evaluation number of each of the runs

is taken into account to compute the new evaluation number, which is always just a continuation of the numbers in the last run. Besides that, the formulas presented in Section 1.2.6 are applied to the data files to filter out rows that do not comply with these rules.

In the end, each problem and instance has its own directory containing the merged data. Note that the data, which is generated by the `Observer` class in a local setting is slightly different. Instead of having one directory per problem id, which includes both the problem number as well as the instance number, there is only one directory per problem that contains the data of multiple instances of the problem. This would have been one extra merge step, but it does not change the `cocopp` output graphs, as I verified empirically. For this reason, I did not include this last step in the merging of the data. The exact procedure that was applied to the `.dat` and `.tdat` files can be seen in Algorithm 2.

Merging the rdat files. In Section 1.2.6 we have seen that the `.rdat` files contain the exact same header as the `.dat` or `.tdat` files. Thus, I copy one of the `.rdat` files for each problem, or I could write a new file by copying the header from either the merged `.dat` or `.tdat` file without further consequences.

Merging the info file I copy one the `.info` files from one of the directories associated with the specific problem and the correct instance in the unmerged data. Then I replace the number of evaluations that were done and the best fitness value that was encountered. To replace the number of evaluations, I copy the evaluation number in the last line of the merged `.tdat` file. The best fitness score can be recorded during the merging of the other data files, then I write it into the `.info` file. In the experiments in this thesis, the suite name in the `.info` file often contained characters that were not part of the utf8 encoding. For this reason, I replaced the suite name in the file with the actual suite identifier as stored in `cocoex.known_suite_names`.

3.3 Experiments

To answer the three research questions presented in Section 3.1, four experiments were designed and conducted:

1. Using a constant number of blocks on a sequence of increasing dimensions on the BBOB suite (`bbob_fnb`)
2. Using a constant block size on a sequence of increasing dimensions on the BBOB suite (`bbob_fbs`)
3. Using a constant number of blocks on a sequence of increasing dimensions on the BBOB-Largescale suite (`bbob_ls_fnb`)
4. Using a constant block size on a sequence of increasing dimensions on the BBOB-Largescale suite (`bbob_ls_fbs`)

The optimization algorithm that was used in all of the experiments was CMA-ES from the `pycma` module described in Section 1.1.1. The initial step-size `sigma0` was set to 2 and the IPOP restart mechanism was used in accordance to Hansen (2019). For the rest of the parameters, the default value was used.

Algorithm 1 Merging the generated `.dat` and `.tdat` files of the `Observers` **within one single run** on a problem. This means that the data is merged across all `BlockWorkers`, i.e. across multiple machines. After the algorithm has been run, every run on a problem instance has its own unique directory.

input
`paths` ▷ Paths to data directories for one problem
`file_extension` ▷ Either `.dat` or `.dat`
`output_directory`
initialization
`runs` ← open files from `paths` and group by runs
procedure MERGE_WITHIN_RUNS
for `run` ∈ `runs` **do**
`files` ← open files of current `run`
`data` ← [lines of 1st file, lines of 2nd file, ...] ▷ Skip headers
`output_lines` ← [Header of first file]
`best_fit` ← ∞ ▷ Minimization
`pointers` ← $[0, \ldots, 0]$
`eval_idx` ← 0
`merged_eval_idx` ← 0
while `data` is not empty **do**
`eval_idx` ← `eval_idx` + 1
`evals` ← list of lines from `data` according to index in `pointers`
`current_min_eval` ← min(`evals`)
if min(`evals`) = `eval_idx` **then**
`current_eval_data` ← `evals` with evaluation number = min(`evals`)
`fits` ← list of fitness scores from `current_eval_data`
`sorted_fit_indices` ← sorted (decreasing) list of indices from `fits`
`num_skipped` ← length(`evals`) - length(`current_eval_data`)
`merged_eval_idx` ← `merged_eval_idx` + `num_skipped`
for i ∈ `sorted_fit_indices` **do**
`merged_eval_idx` ← `merged_eval_idx` + 1
`fit` ← `fits`[i]
if `fit` < `best_fit` **or** `file_extension` = ".tdat" **then**
`best_fit` ← min{`fit`, `best_fit`}
`line` ← `current_eval_data`[i]
Change evaluation number in `line` to `merged_eval_idx`
Append `line` to `output_lines`
Increment value in `pointers` that correspond to the file from which `fit`[i] was read
if `pointers`[i] ≥ length of i-th file **then**
Flag `data`[i] for removal
Delete elements from `data` that were flagged for removal
else
`merged_eval_idx` ← `merged_eval_idx` + length(`data`)
Write data file with extension `file_extension` to `output_directory`

Algorithm 2 Merge the generated `.dat`, `.tdat`, and `.info` files of the `Observer` **across multiple runs** on a problem. In the end, the data of each problem instance has its own unique directory.

input
  `experiment_directory` ▷ Path to the directory containing the experiment data
  `file_extension` ▷ Either `dat` or `dat`
initialization
  `runs_per_problem` ← group paths in `experiment_directory` by problems
procedure MERGE_WITHIN_RUNS
  for `runs` ∈ `runs_per_problem` **do**
    if length(`runs`) = 1 **then**
      Trim "_run" suffix from directory name in `runs`[0]
      Continue with next problem
    offsets = [0]
    for $i = 0, \ldots,$ length(`runs`) $- 2$ **do** ▷ Skip the last element
      Append last evaluation number in info file to `offsets` ▷ Header only once
    `best_fit` = ∞
    `output_lines` ← Read files in `runs` and concatenate lines
    $k \leftarrow 999$ ▷ Current exponent in tdat formula $10^{k/5}$
    `last_eval_num` ← 0
    `run_number` ← 1
    for $i = 1, \ldots,$ length(`output_lines`) **do**
      `eval_num` ← Extract evaluation number from i-th line in `output_lines`
      `fit` ← Extract fitness value from i-th line in `output_lines`
      if `eval_num` $<$ `last_eval_num` **then**
        `run_number` ← `run_number` + 1
      `last_eval_num` ← `eval_num`
      `offset` ← Sum of the first `run_number` values in `offsets`
      if `file_extension`="dat" **and** `fit` $\geq 10^{k/5}$ **then**
        `output_lines`[i] ← null ▷ Mark for deletion
        continue with next line
      if `file_extension`="tdat" **and** $\exists \ell \in \mathbb{N}$ s.t. $i = \left\lfloor 10^{\ell/20} \right\rfloor$ **then**
        `new_eval_num` ← `eval_num` + `offset`
        if $\nexists \ell \in \mathbb{N}$ s.t. $i = \left\lfloor 10^{\ell/20} \right\rfloor$ **then**
          `output_lines`[i] ← null ▷ Mark for deletion
          Continue with next line
        `best_fit` ← min{`fit`, `best_fit`}
        if `file_extension`="dat" **then**
          $k \leftarrow \lceil 5 \cdot \log_{10}$ `fit` $\rceil$ ▷ Update last seen exponent
        Add `offset` to evaluation number in i-th line
        Remove lines from `output_lines` that were flagged for deletion
        `last_eval_num` ← last evaluation number in `output_lines`
    Write `output_lines` to path in `runs`[0]
    if `best_fit` $<$ fitness recorded in info file **then**
      Replace fitness value in info file with `best_fit`
    Trim "_run" suffix from directory name in `runs`[0]

The details of the experiments can be seen in Tables 3.1 to 3.4. For the experiments with a fixed number of blocks (`fnb`), the dimensionality of each block increased with growing dimensionality of the problem. On the other hand, for the experiments with a fixed block size (`fbs`), the number of blocks increased with growing dimensionality of the problem. I used two blocks for the experiment `bbob_fbs`, and sixteen for experiment `bbob_ls_fbs`. For experiments `bbob_fbs` and `bbob_ls_fbs` I chose a block size of 5 and 40 respectively. Because of time limitations, a budget multiplier of 10^4 was used for every experiment. Therefore, in the largest (640d) suite, each problem was evaluated at most $640 \cdot 10^4 = 6.4 \cdot 10^6$ times. Even though DiBB is asynchronous, we can estimate the number of evaluations per machine under the assumption that the evaluations are distributed uniformly across all `BlockWorkers`. In the case of the 640d suite this means that each machine performs approximately 400 000 evaluations.

Number of dimensions	Number of blocks	Block size	Experiment ID	Evaluations per machine
5	2	2, 3	`bbob_fnb_05d_2b`	25.000
10	2	5	`bbob_fnb_10d_2b`	50.000
20	2	10	`bbob_fnb_20d_2b`	100.000
40	2	20	`bbob_fnb_40d_2b`	200.000

TABLE 3.1: Experiment 1 on the BBOB suite with a fixed number of blocks (`bbob_fnb`). The more dimensions, the larger the block size, which means that each block has to perform more evaluations. Larger blocks also mean that more correlations are taken into account.

Number of dimensions	Number of blocks	Block size	Experiment ID	Evaluations per machine
5	1	5	`bbob_fbs_05d_1b`	50.000
10	2	5	`bbob_fbs_10d_2b`	50.000
20	4	5	`bbob_fbs_20d_4b`	50.000
40	8	5	`bbob_fbs_40d_8b`	50.000

TABLE 3.2: Experiment 2 on the BBOB suite with a fixed block size `bbob_fbs`. The more dimensions, the more blocks are needed. The number of evaluations per machine is the same. The number of correlations that are taken into account in each block is constant.

3.4 Results

3.4.1 Experiment 1

The ECDF plot of the experiment `bbob_fnb` on the BBOB suite and over all functions and all instances can be seen in Figure 3.1. The best performance was achieved in the problems of dimension 5. With each increase in the dimensionality, the two blocks converge to a solution that is farther away from the optimal solution. It is important to note that this is not 100% confident, because the curves after the cross are using the *simulated restarts* as explained in Section 1.2.5. Indeed, we can see that the distance between the curves is smaller before `cocopp` started using simulated

Number of dimensions	Number of blocks	Block size	Experiment ID	Evaluations per machine
80	16	5	`bbob_ls_fnb_080d_16b`	50.000
160	16	10	`bbob_ls_fnb_160d_16b`	100.000
320	16	20	`bbob_ls_fnb_320d_16b`	200.000
640	16	40	`bbob_ls_fnb_640d_16b`	400.000

TABLE 3.3: Experiment 3 on the BBOB-Largescale suite with a fixed number of blocks (`bbob_ls_fnb`). The more dimensions, the larger the block size, which means that each block has to perform more evaluations. Larger blocks also mean that more correlations are taken into account.

Number of dimensions	Number of blocks	Block size	Experiment ID	Evaluations per machine
40	1	40	`bbob_ls_fbs_040d_01b`	400.000
80	2	40	`bbob_ls_fbs_080d_02b`	400.000
160	4	40	`bbob_ls_fbs_160d_04b`	400.000
320	8	40	`bbob_ls_fbs_320d_08b`	400.000
640	16	40	`bbob_ls_fbs_640d_16b`	400.000

TABLE 3.4: Experiment 4 on the BBOB-Largescale suite with a fixed block size (`bbob_ls_fbs`). The more dimensions, the more blocks are needed. The number of evaluations per machine is the same. The number of correlations that are taken into account in each block is constant.

restarts to predict the outcome of the optimization. Nonetheless, it is normal that problems with a larger dimension are harder to optimize.

Figure 3.2 to Figure 3.4 show the ECDF plots for all 24 functions. There are some interesting observations. First, we can see that the performance is good on the first two separable problems. This was to be expected, since the parameters of a separable functions are independent. For the Rastrigin (f3 and f15; see Figures 3.2c and 3.3g), the performance is good in dimensions 5 and 10. For dimensions 20 and 40, the algorithm converges towards solutions that are worse. These observations were anticipated for the non-separable version of the Rastrigin function (f15), since a larger dimensionality leads to a more complex search space. In this case, the function has roughly 10^d local optima (Hansen, Finck, et al., 2009), where d is the dimensionality of the problem. For the separable Rastrigin function (f3), the worse performance is most likely also due to the higher dimensionality. This means that 20 dimensions or more make the Rastrigin function too hard for CMA-ES to solve without any further hyperparameter tuning, even though it is separable[1].

For the Rosenbrock function (f8; see Figure 3.2h), we have good results from dimension 10 and upwards. In five dimensions, the performance is much worse, which could be due to the smaller block size that results in less correlations being considered. This also shows that there is a critical number of correlations that need to be considered in certain problems in order to find a good solution.

[1]The same behavior can also be seen in the latest CMA-ES baseline data from Hansen (2019), which can be accessed at http://lq-cma.gforge.inria.fr/ppdata-archives/pap-gecco2019/CMA-E_lq-CM/pprldmany-single-functions/pprldmany.html

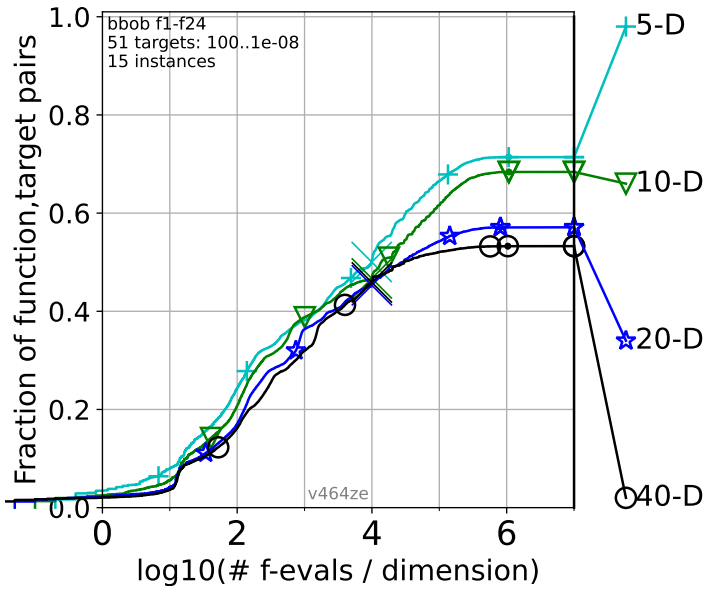


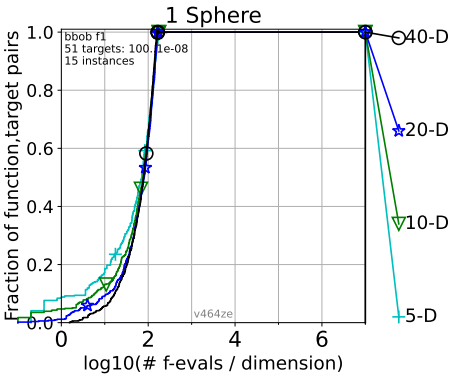
FIGURE 3.1: ECDF graph showing the performance of DiBB with CMA-ES and 2 blocks on the BBOB suite for different dimensions across all 24 problems. With increasing dimensionality, the performance gets worse.

For the Schwefel function (f20; see Figure 3.4d), the algorithm only converges to good solutions for dimensions 5 and 10. For dimensions 20 and 40, the optimization seems to be much harder. Similarly, the Gallagher 21 peaks (f22; see Figure 3.4f) only converges to the optimal value for dimensions 5, 10, and 20. For dimension 40, the algorithm converges to a worse solutions. These observations are in line with the observations of the Rastrigin function: larger dimensions lead to a more complex search space, which makes it harder to solve the problem, independent on the number of blocks that are being used.

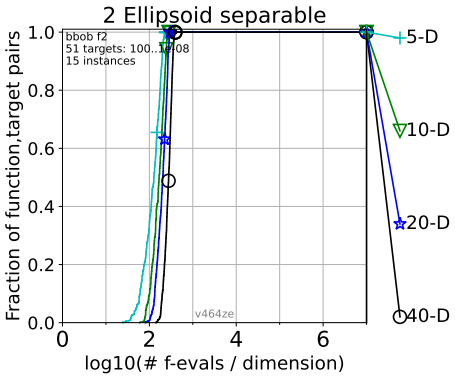
3.4.2 Experiment 2

In this experiment, BBOB problems of different dimensions were evaluated with a constant block size rather than a constant number of blocks. Figure 3.5 compares the results of experiment 1 (`bbob_fnb`) with the ones from experiment 2 (`bbob_fbs`). We can see that the curves corresponding to larger dimensions in experiment `bbob_fbs` (Figure 3.5b) are farther away from each other than the ones in experiment `bbob_fbn`. Because the block size in experiment `bbob_fbs` was fixed, the number of blocks increased with growing dimensionality. On the other hand, we have a fixed number of blocks in experiment `bbob_fnb`, which leads to larger blocks for larger dimensionalities. However, the change in dimensionality of the blocks does not affect the performance of the optimizer to the same extend as the number of blocks, as Figure 3.5a shows. Overall, the performance of CMA-ES in experiment `bbob_fnb` seems to be better than the performance in experiment `bbob_fbs`, except for the 5-dimensional case.

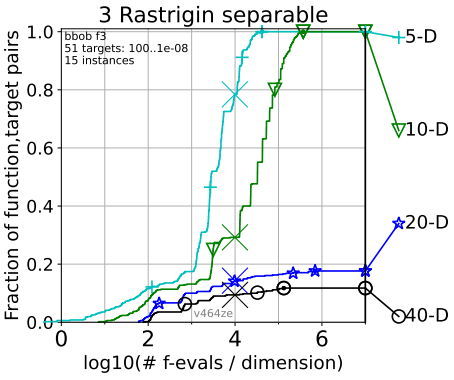
This difference in performance can be explained by the sizes of the blocks: in experiment `bbob_fnb`, the 10d suite uses the same block size as the one in experiment `bbob_fbs`, and the 20d and 40d suites use larger block sizes. The lower performance of the 5d suite in experiment `bbob_fbn` can be explained similarly. We also see that the difference in performance for the block sizes in the 10d and 20d suites is smaller



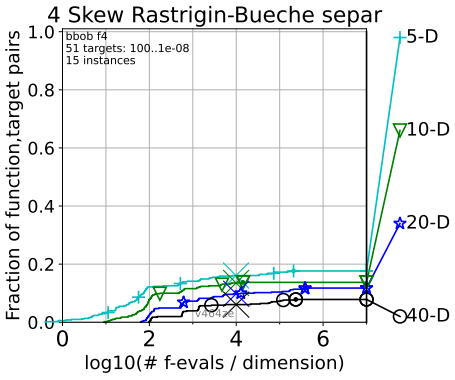
(A) Sphere function



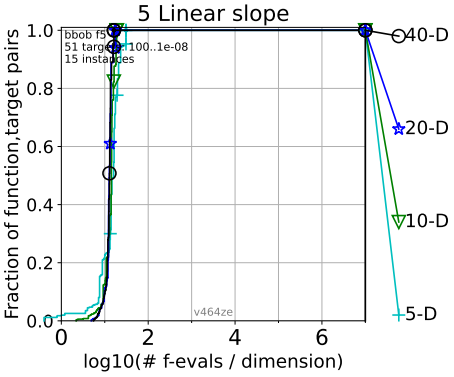
(B) Ellipsoidal function



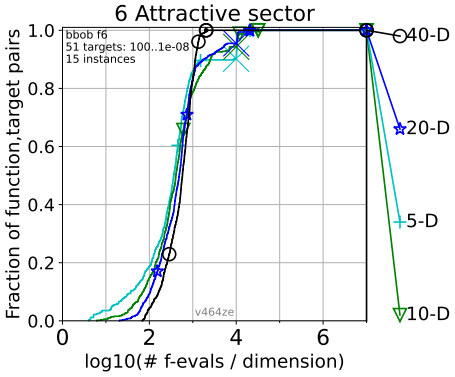
(C) Rastrigin function



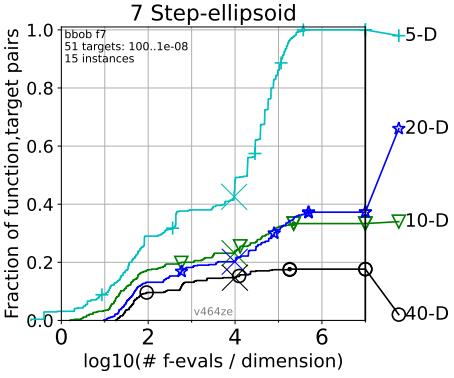
(D) Büche-Rastrigin function



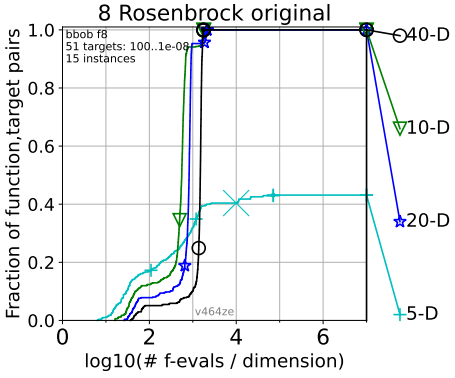
(E) Linear slope



(F) Attractive Sector function

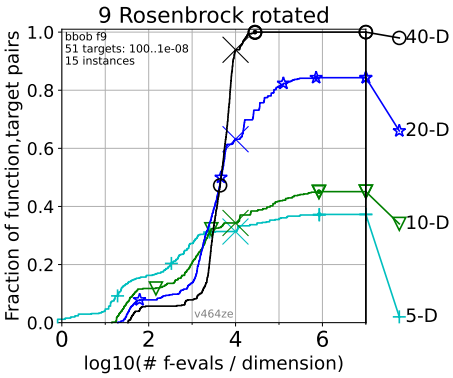


(G) Step-Ellipsoidal function

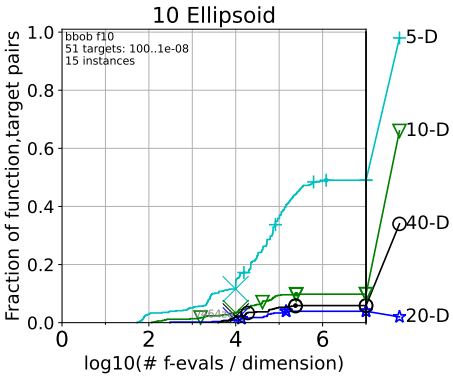


(H) Rosenbrock function, original

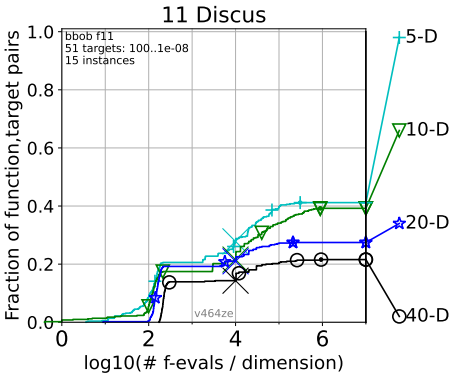
FIGURE 3.2: ECDF plots for functions f1 to f8 in the BBOB suite, aggregated across all instances of each function. The experiment used suites of dimensions 5, 10, 20, and 40 with two blocks each. Thus, the larger the dimension of a suite of problems, the larger the blocks that were used within DiBB.



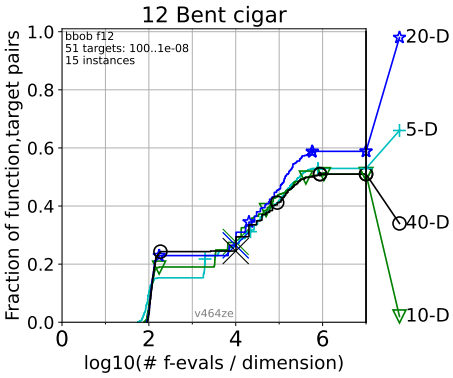
(A) Rosenbrock function, rotated



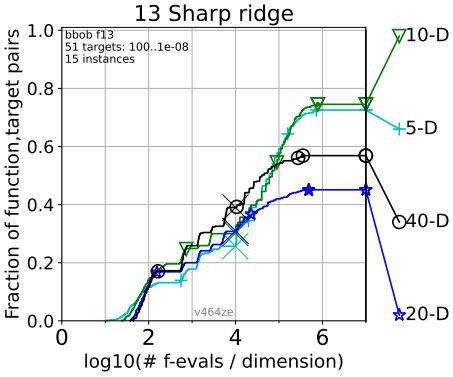
(B) Ellipsoidal function



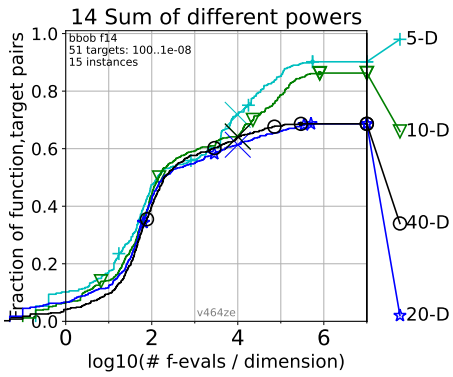
(C) Discus function



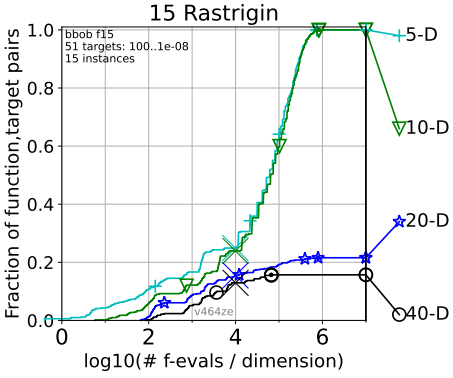
(D) Bent Cigar function



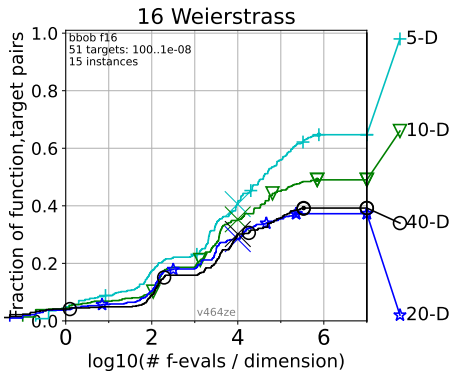
(E) Sharp Ridge function



(F) Different Powers function

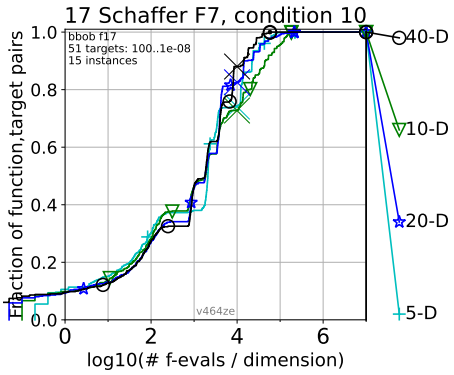


(G) Rastrigin function

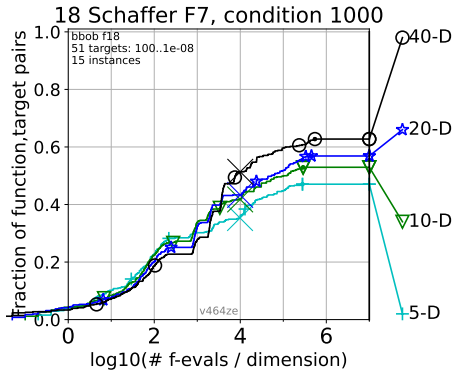


(H) Weierstrass function

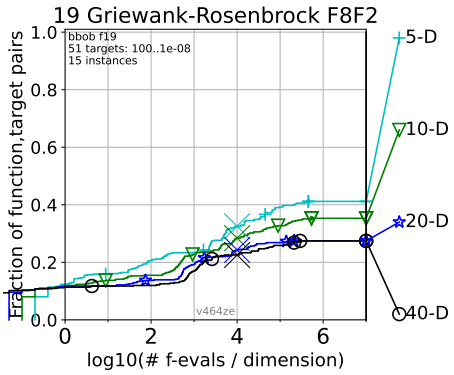
FIGURE 3.3: ECDF plots for functions f9 to f16 in the BBOB suite, aggregated across all instances of each function. The experiment used suites of dimensions 5, 10, 20, and 40 with two blocks each. Thus, the larger the dimension of a suite of problems, the larger the blocks that were used within DiBB.



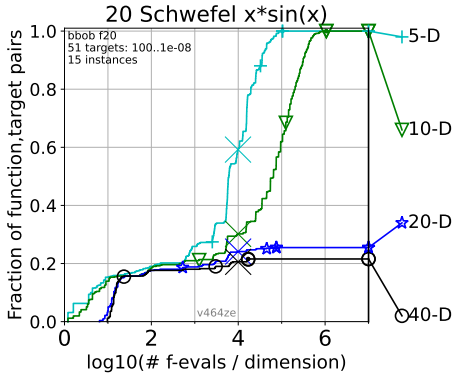
(A) Schaffers F7 function



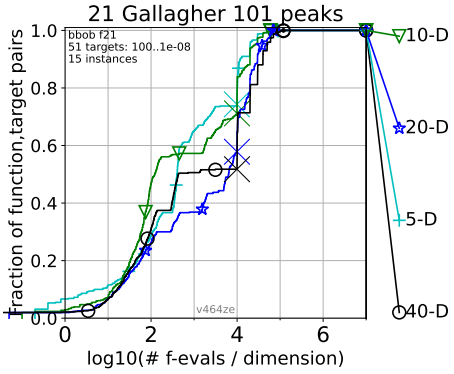
(B) Schaffers F7 function (moderately ill-conditioned)



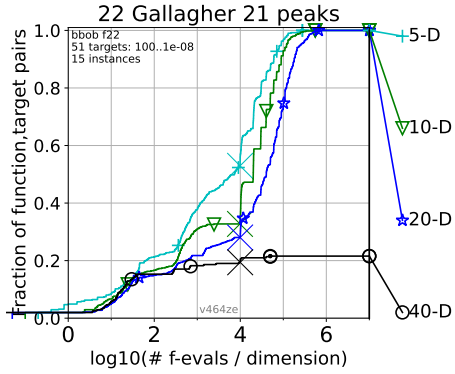
(C) Composite Griewank-Rosenbrock function F8F2



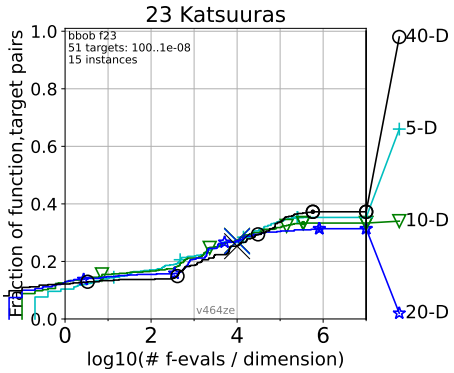
(D) Schwefel function



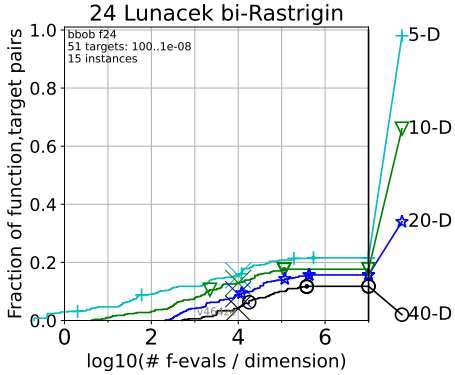
(E) Gallagher's Gaussian 101-me Peaks function



(F) Gallagher's Gaussian 21-hi Peaks function

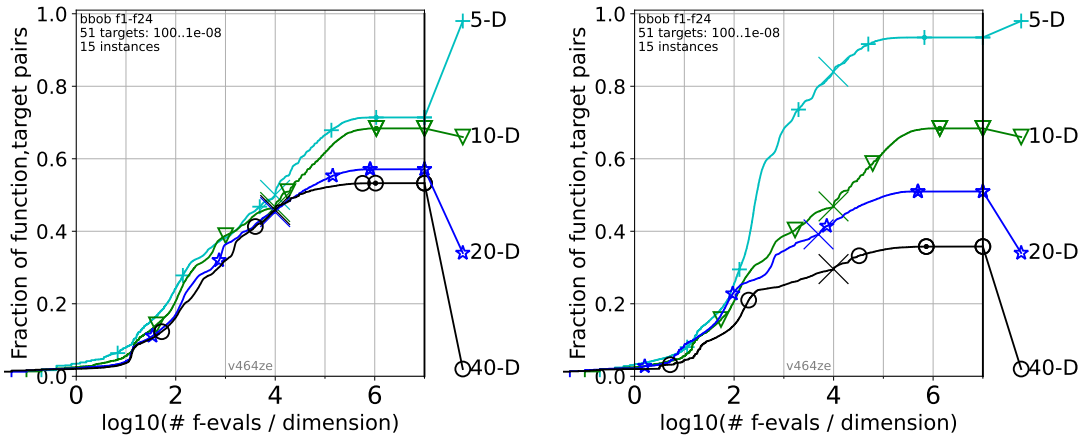


(G) Katsuura function



(H) Lunacek bi-Rastrigin function

FIGURE 3.4: ECDF plots for functions f17 to f24 in the BBOB suite, aggregated across all instances of each function. The experiment used suites of dimensions 5, 10, 20, and 40 with two blocks each. Thus, the larger the dimension of a suite of problems, the larger the blocks that were used within DiBB.



(A) ECDF graph showing the performance of DiBB with CMA-ES and 2 blocks on the BBOB suite for different dimensions across all 24 problems. With increasing dimensionality, the performance gets worse.

(B) ECDF graph showing the performance of DiBB with CMA-ES and blocks of size 5 on the BBOB suite for different dimensions. With increasing dimensionality, the performance gets worse.

FIGURE 3.5: Comparison of the ECDF plots over all 24 functions of experiments `bbob_fnb` (left) and `bbob_fbs` (right).

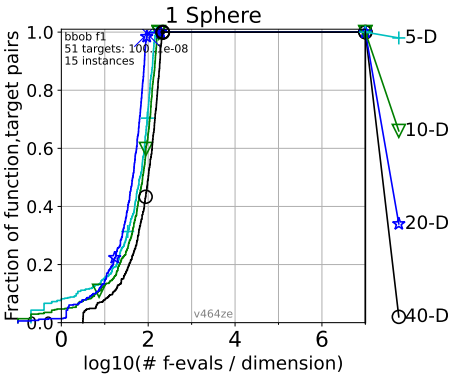
than the one in the 40d suite. A block size of 20 (Figure 3.5a) seems to capture significantly more meaningful information about the problem than a block size of 5 (Figure 3.5b) on the 40d suite. This shows that tuning the individual block sizes of DiBB for a certain problem is likely to yield better performance than to strictly adhere to a fixed number of blocks.

Figures 3.6 to 3.8 show the ECDF plots for all 24 problems. The suite with five dimensions has the best performance, which is expected: First, a smaller dimension usually also means less local optima; Second, we are using blocks of size 5. In five dimensions, this means that we are using only one block and thus the entire correlational information (same as CMA-ES). Especially in the group of ill-conditioned functions (f10-f14; see Figures 3.7b to 3.7f), the full covariance information seems to significantly improve performance. For the Rastrigin (f3 and f15; see Figures 3.6c and 3.7g), we see the same behavior as in the first experiment: the optimizer is able to find the optimal solution in dimensions 5 and 10, however the performance is much worse in dimensions 20 and 40. The same happens for the Schwefel function (f20; see Figure 3.8d). The Ellipsoid (f10; see Figure 3.7b) cannot be solved in dimensions 20 and 40 at all, which is most likely because of a too small block size. Interestingly, the Gallagher function with 101 peaks (f21; see Figure 3.8e) has been easier to solve in 40 dimensions than the Gallagher function with 21 peaks.

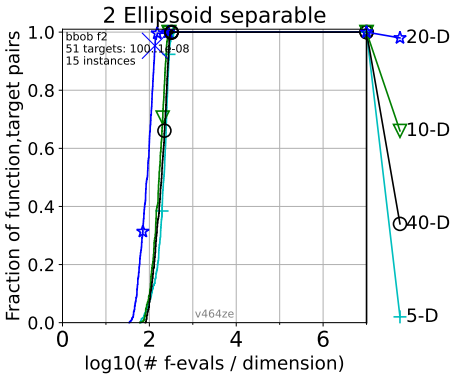
3.4.3 Experiment 3

For the experiment `bbob_ls_fnb`, the ECDF plot over all functions and instances looks surprisingly similar in all dimensions. The plot can be seen in Figure 3.9. If we look at the plots in Figures 3.10 to 3.12, where the performance on all 24 functions is shown, we can see that most of the curves in the graphs do not differ notably.

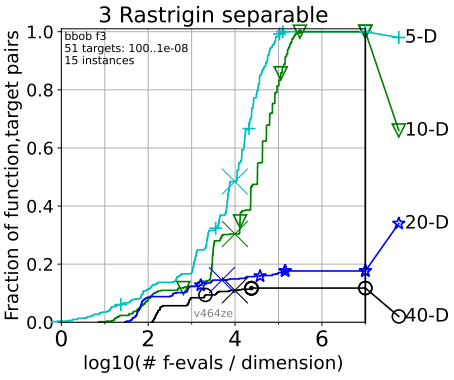
However, there are some exceptions. First, there is a difference in the Bent Cigar (f12; see Figure 3.11d), which follows the expected pattern, where a problem with less dimensions comes closer to the optimal value than a problem with more dimensions. The same is true for the Schaffer F7 problem with a condition number of 10 (f17; see Figure 3.12a) or 1000 (f18; see Figure 3.12b), and the Schwefel function (f20;



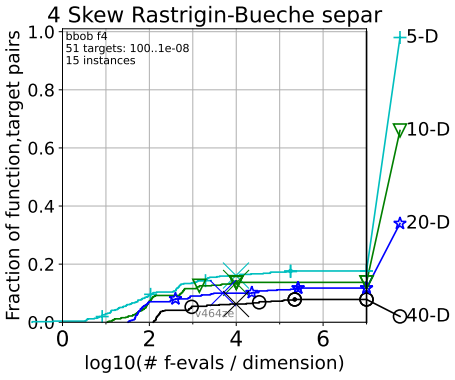
(A) Sphere function



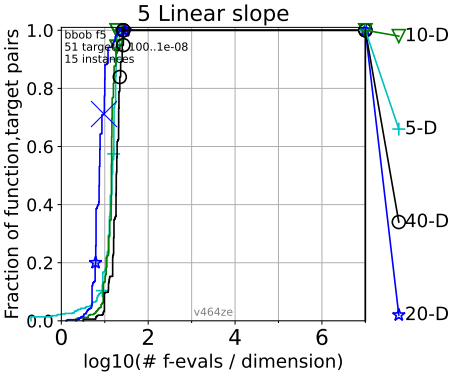
(B) Ellipsoidal function



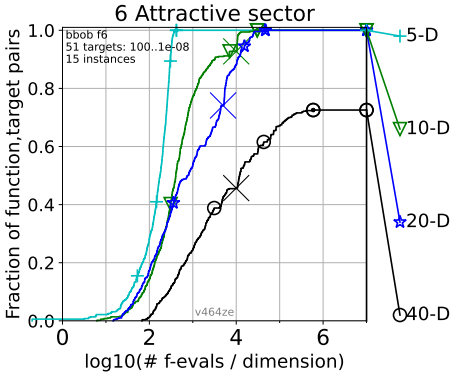
(C) Rastrigin function



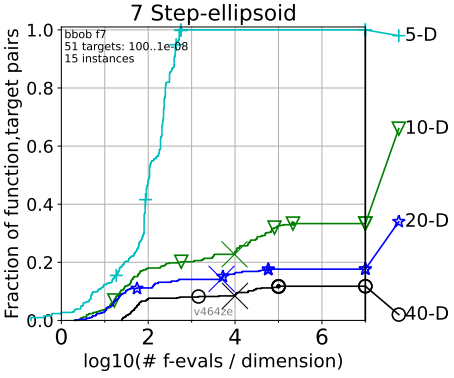
(D) Büche-Rastrigin function



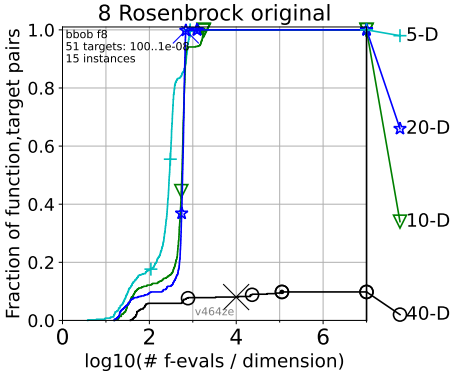
(E) Linear slope



(F) Attractive Sector function

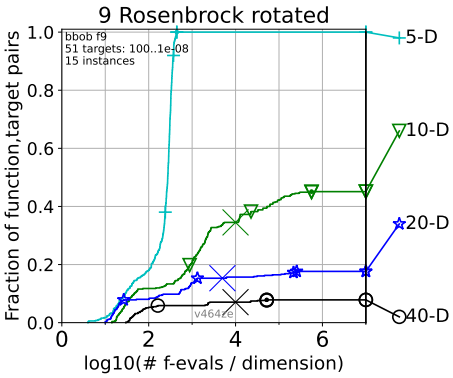


(G) Step-Ellipsoidal function

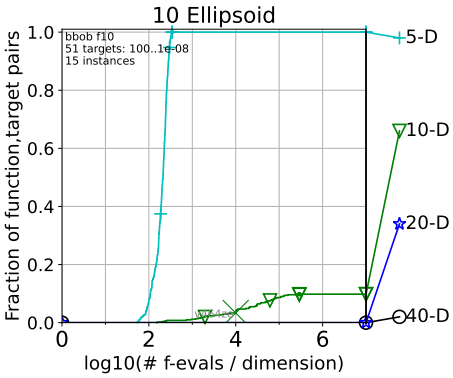


(H) Rosenbrock function, original

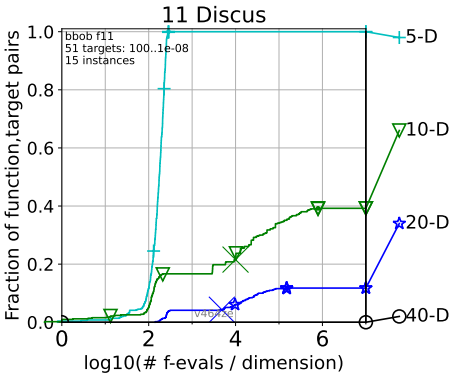
FIGURE 3.6: ECDF plots for functions f1 to f8 in the BBOB suite, aggregated across all instances of each function. The experiment used suites of dimensions 5, 10, 20, and 40 with blocks of size 5 each. Thus, the larger the dimension of a suite of problems, the more blocks were used within DiBB.



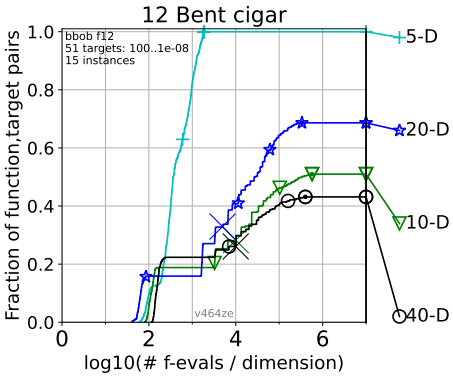
(A) Rosenbrock function, rotated



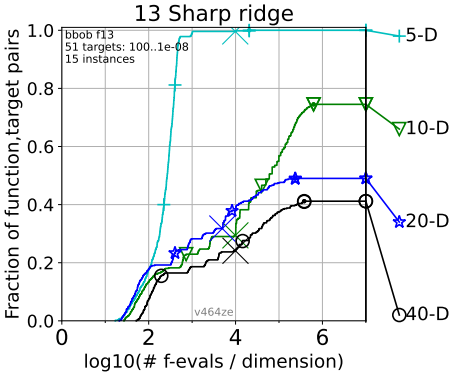
(B) Ellipsoidal function



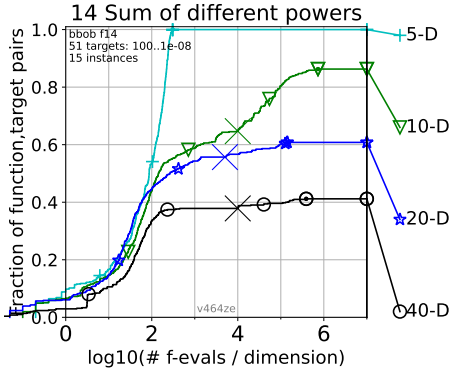
(C) Discus function



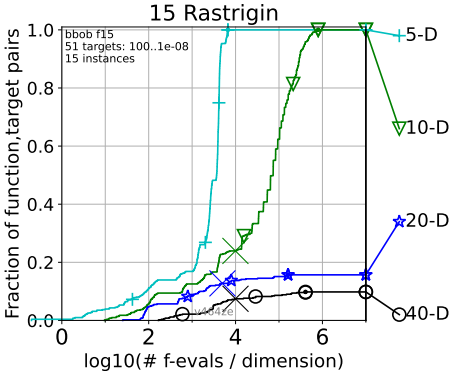
(D) Bent Cigar function



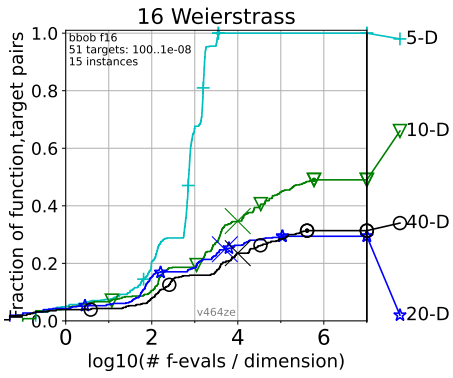
(E) Sharp Ridge function



(F) Different Powers function

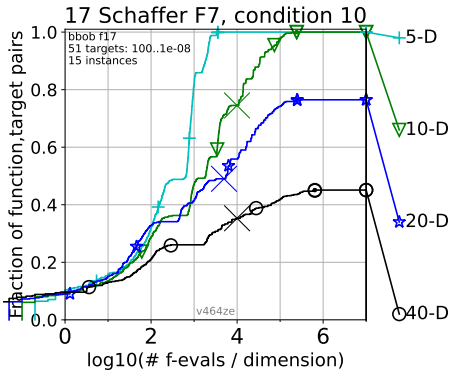


(G) Rastrigin function

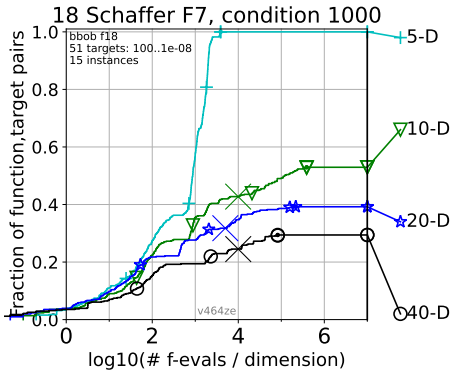


(H) Weierstrass function

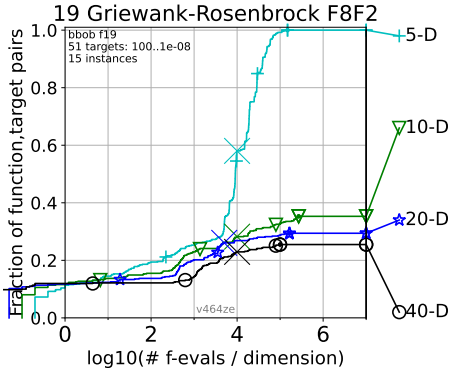
FIGURE 3.7: ECDF plots for functions f9 to f16 in the BBOB suite, aggregated across all instances of each function. The experiment used suites of dimensions 5, 10, 20, and 40 with blocks of size 5 each. Thus, the larger the dimension of a suite of problems, the more blocks were used within DiBB.



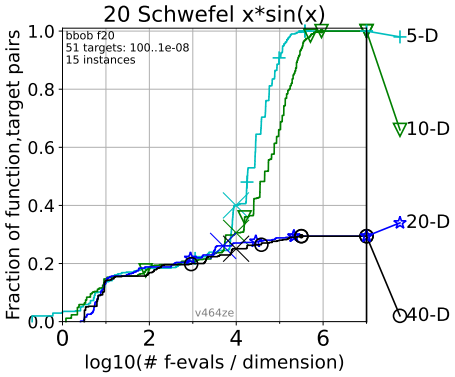
(A) Schaffers F7 function



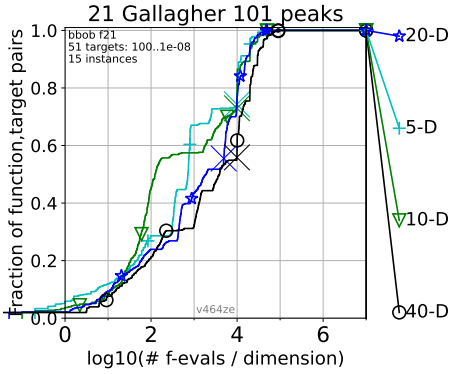
(B) Schaffers F7 function (moderately ill-conditioned)



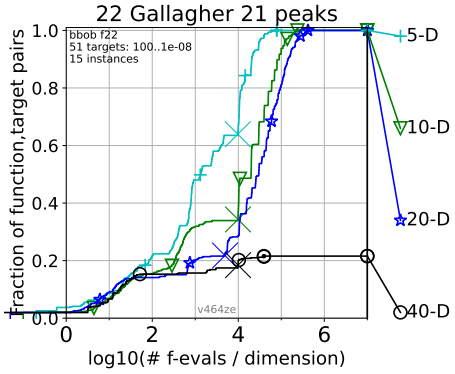
(C) Composite Griewank-Rosenbrock function F8F2



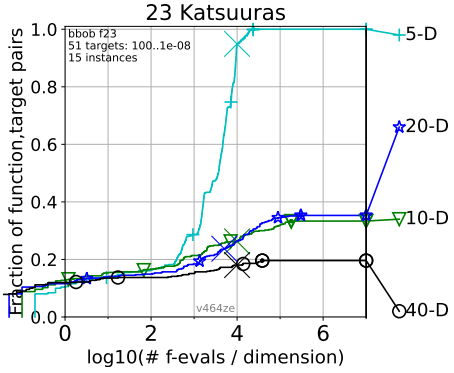
(D) Schwefel function



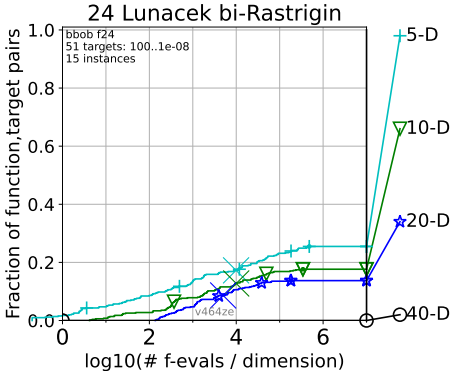
(E) Gallagher's Gaussian 101-me Peaks function



(F) Gallagher's Gaussian 21-hi Peaks function



(G) Katsuura function



(H) Lunacek bi-Rastrigin function

FIGURE 3.8: ECDF plots for functions f17 to f24 in the BBOB suite, aggregated across all instances of each function. The experiment used suites of dimensions 5, 10, 20, and 40 with blocks of size 5 each. Thus, the larger the dimension of a suite of problems, the more blocks were used within DiBB.

see Figure 3.12d). Interestingly, there are also problems where the opposite occurs. For the sum of different powers (f14; see Figure 3.11f) and Katsuuras (f23; see Figure 3.12g), the solution that was found during optimization is closer to the optimal value if the dimension of the problem is larger.

This may seem counter-intuitive because larger dimensions should make it harder for a problem to be optimized. However, it can be explained by taking the block size into account. For a problem with 80 dimensions, we get blocks of size 5 if we use 16 blocks. However, if we use 16 blocks on a problem with 640 dimensions, we have a block size of 40 dimensions. This tells us that a block size of 5 is most likely less informative for problems f14 and f23. It is to be expected that a larger block size would lead to an even better performance. Nevertheless, a larger block size also means that CMA-ES has to do more computations. Because they do not scale linearly but quadratically with the dimension of the covariance matrix, larger block sizes can come at a high performance cost.

We should also take a look at the Gallagher 101 peaks (f21; see Figure 3.12e). In this case, the order of the dimensions which could be solved more accurately is 320, 160, 640, and 80. At first, this order seems confusing. However, as we have seen for problems f14 and f23, it can give us information about the performance of different block sizes. A block size of 40 dimensions, which corresponds to the problems of dimension 640 in the plots, lowers performance significantly. In this case, a block size of 20 outperforms all other block sizes that were tried. This underlines the importance of experimenting with different block sizes before using DiBB if the blocks of parameter where intrinsic correlation is not known in advance.

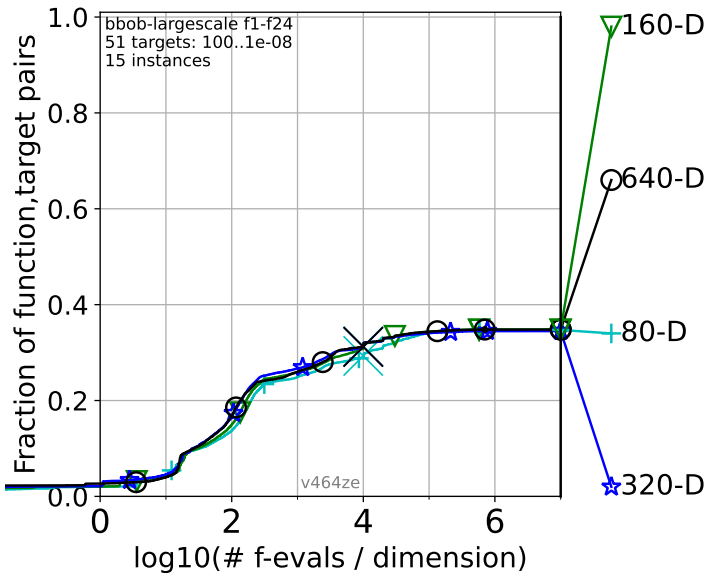
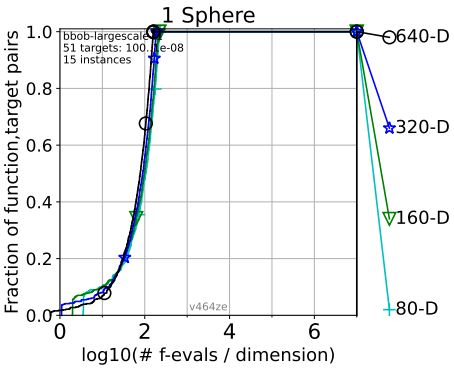


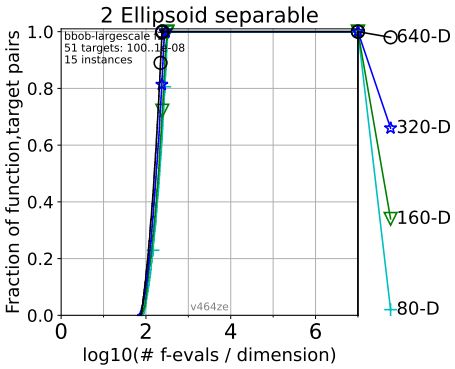
FIGURE 3.9: ECDF graph showing the performance of DiBB with CMA-ES and 16 blocks on the BBOB-Largescale suite for different dimensions. In contrast to the BBOB experiment, the change in dimension is not reflected in the performance of the algorithm anymore.

3.4.4 Experiment 4

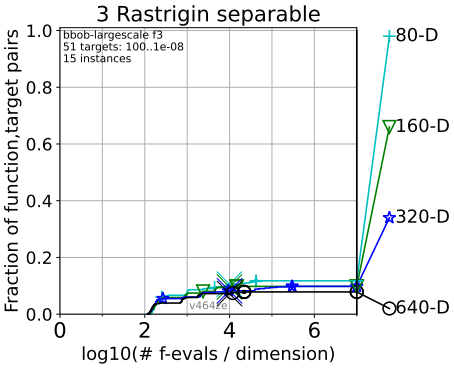
For this experiment, I chose a constant block size of 40. In Figure 3.13, we can see that the some of the curves are closer than others, which does not resemble the ECDF plot of the same experiment on the BBOB suite that we saw earlier. Particularly, the



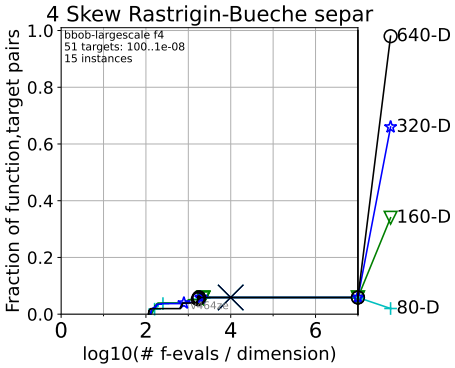
(A) Sphere function



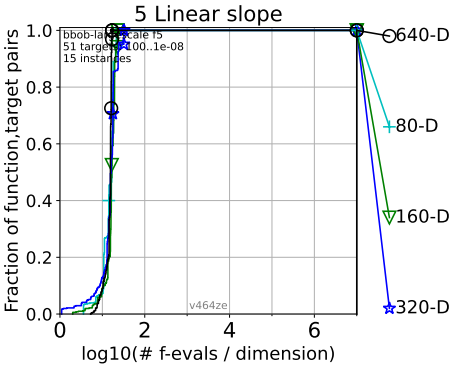
(B) Ellipsoidal function



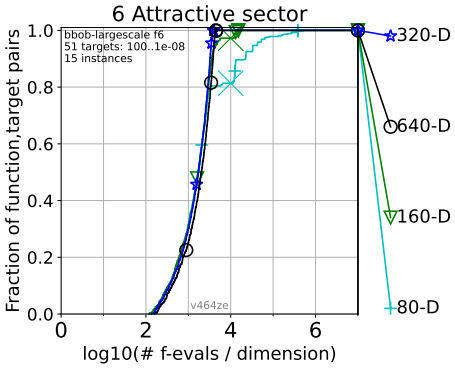
(C) Rastrigin function



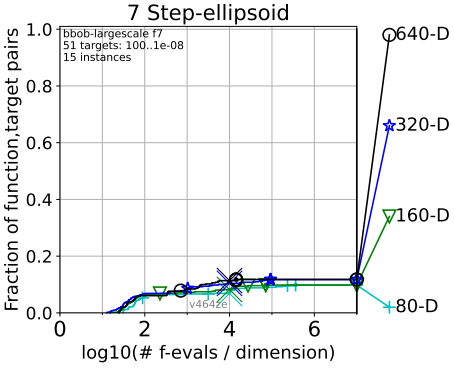
(D) Büche-Rastrigin function



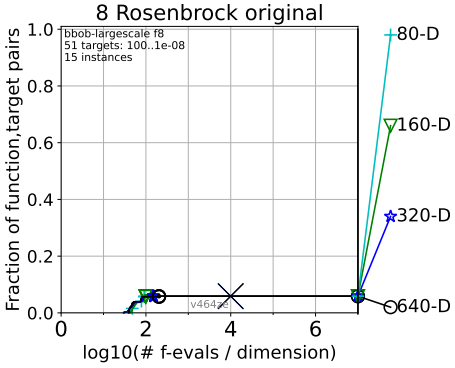
(E) Linear slope



(F) Attractive Sector function

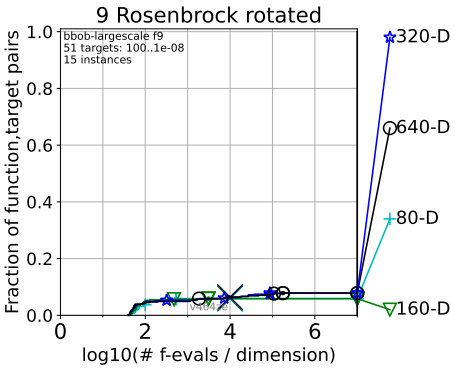


(G) Step-Ellipsoidal function

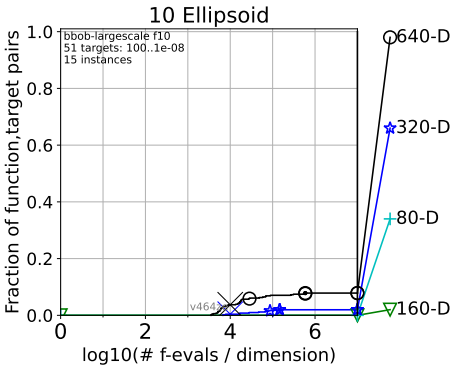


(H) Rosenbrock function, original

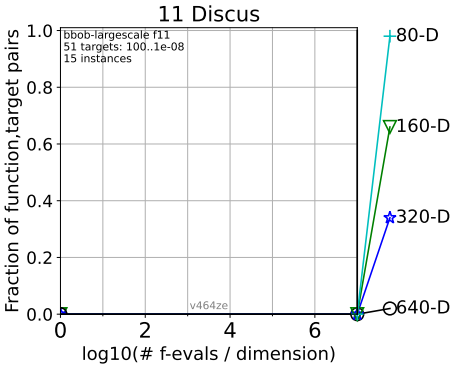
FIGURE 3.10: ECDF plots for functions f1 to f8 in the BBOB-Largescale suite, aggregated across all instances of each function. The experiment used suites of dimensions 80, 160, 320, and 640 with 16 blocks each. Thus, the larger the dimension of a suite of problems, the larger the blocks that were used within DiBB.



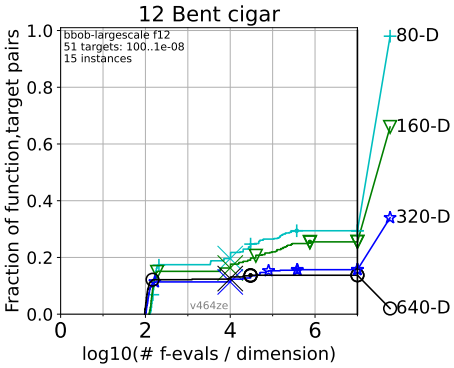
(A) Rosenbrock function, rotated



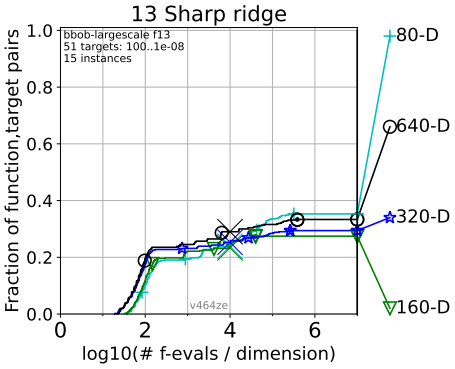
(B) Ellipsoidal function



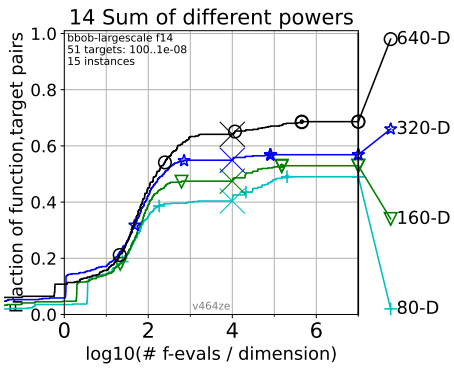
(C) Discus function



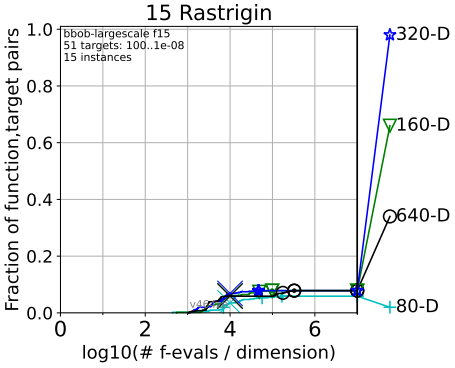
(D) Bent Cigar function



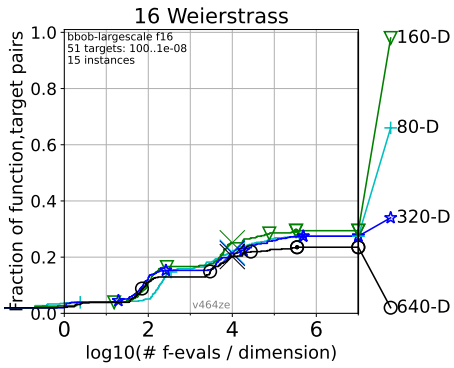
(E) Sharp Ridge function



(F) Different Powers function

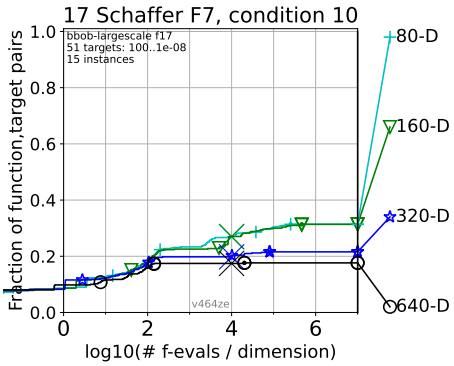


(G) Rastrigin function

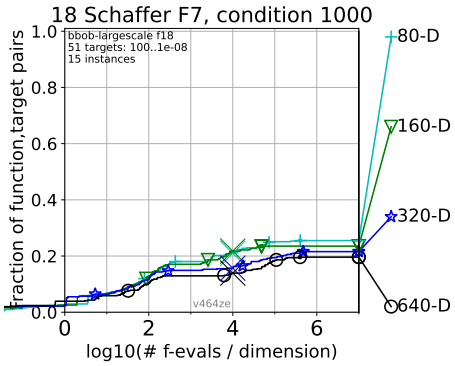


(H) Weierstrass function

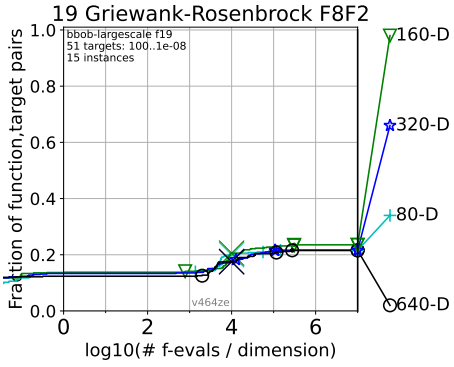
FIGURE 3.11: ECDF plots for functions f9 to 16 in the BBOB-Largescale suite, aggregated across all instances of each function. The experiment used suites of dimensions 80, 160, 320, and 640 with 16 blocks each. Thus, the larger the dimension of a suite of problems, the larger the blocks that were used within DiBB.



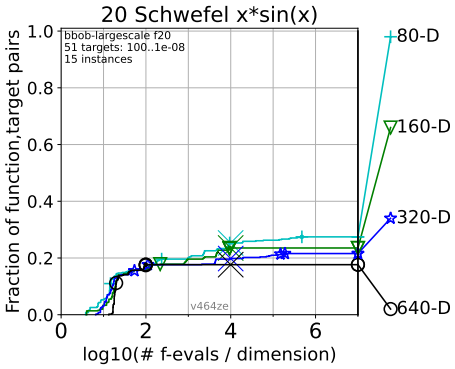
(A) Schaffers F7 function



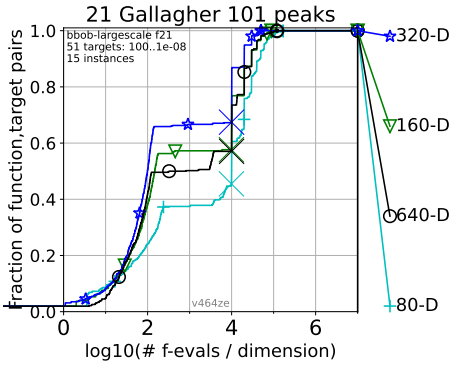
(B) Schaffers F7 function (moderately ill-conditioned)



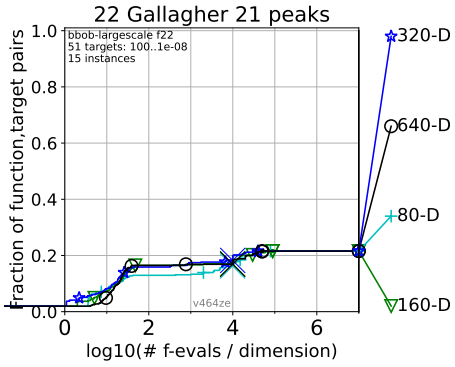
(C) Composite Griewank-Rosenbrock function F8F2



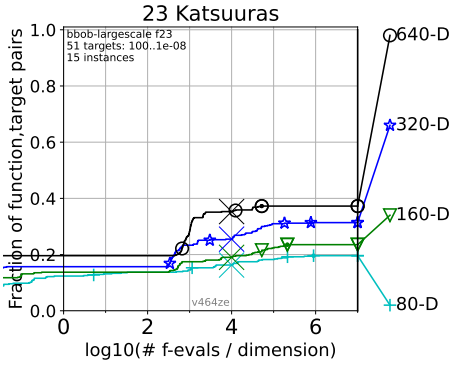
(D) Schwefel function



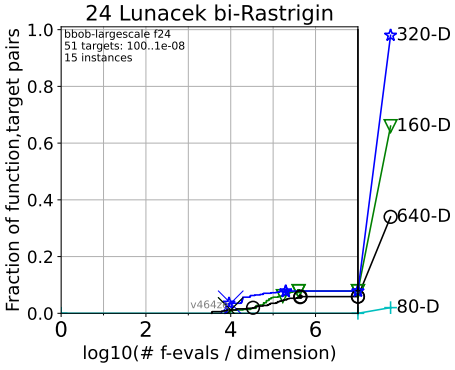
(E) Gallagher's Gaussian 101-me Peaks function



(F) Gallagher's Gaussian 21-hi Peaks function



(G) Katsuura function



(H) Lunacek bi-Rastrigin function

FIGURE 3.12: ECDF plots for functions f17 to f24 in the BBOB-Largescale suite, aggregated across all instances of each function. The experiment used suites of dimensions 80, 160, 320, and 640 with 16 blocks each. Thus, the larger the dimension of a suite of problems, the larger the blocks that were used within DiBB.

curves for problems of dimension 160, 320 and 640 are close at the point where the simulated restarts begin. Moreover, when comparing the graph with the one in Figure 3.9, we can see that CMA-ES found better solutions for the problems in dimensions 40 and 80. For the 40-dimensional problems, this was to be expected, since it only uses one block and thus the entire correlational information. The increased performance in the problems of dimension 80 is also anticipated, since two larger blocks maintain significantly more correlational information than 16 smaller ones. Therefore, we can say that using a block size of 5 is not optimal on the BBOB-Largescale problems in dimension 80.

We also see that the performance between dimensions 320 and 640 decreases only slightly. Because the performance loss caused by doubling the number of blocks is only small, we can essentially tackle problem twice as large as the 320d problems and still get a similar performance. Therefore, blocks with a covariance matrix $\mathbf{C} \in \mathbb{R}^{40\times40}$ seem to be sufficient to solve both the 320d and the 640d BBOB-Largescale suite with similar accuracy. In Section 3.4.6 we will see that not only the performance but also the wall-clock speed of the experiments on the 320d and the 640d suites are comparable.

Figures 3.14 to 3.16 show that the performance of CMA-ES with blocks of size 40 is nearly the same on all the dimensions for functions f1 to f6 (Figures 3.14a to 3.14f), f20 (Figure 3.16d), f22 (Figure 3.16f), and f24 (Figure 3.16h). While this had to be expected on the group of separable problems (f1 to f5), it is surprising for the non-separable functions. This means that CMA-ES did not benefit from the entire correlational information that was available on the 40d problems on these functions. If we look at the Gallagher 101 peaks (f21; see Figure 3.16e), we can see that the best performance was achieved on problems of dimension 320. However, on the BBOB suite with a fixed block size of 5 (`bbob_fbs`), the lower dimensions had a better performance than the higher ones. Thus, a block size of 40 seems to be a good fit for this specific function.

3.4.5 Comparing DiBB to Other Methods

Especially on larger dimensions, it is important to look at the performance of DiBB with CMA-ES compared to a standard CMA-ES algorithm that uses the entire covariance matrix and a the separable version of CMA-ES that uses only diagonal covariance matrices, i.e. the exact same number of blocks than the number of dimensions. For this reason, I ran experiment `bbob_ls_fbs` again, but this time without using the IPOP restart strategy to match the experimental setup described by Varelas (2019). Then, I generated the ECDF plots to compare DiBB the CMA[2] and sepCMA data[3] provided by Varelas. Note that I am not re-rerunning any of Varelas' experiments. I am only using the data generated during his experiments. Figure 3.17 shows the ECDF plots.

We can see that the CMA baseline always outperforms both DiBB with CMA-ES and the separable CMA-ES variant with a diagonal covariance matrix, which was expected. For dimension 80 and two blocks, the DiBB outperforms sepCMA. In dimension 160, sepCMA achieved a slightly better performance before the beginning of the simulated restarts. In dimension 320, DiBB was slightly better than sepCMA before the simulated restarts were used to predict the performance for larger budget

[2]The data can be downloaded via `cocopp` or the following link: https://coco.gforge.inria.fr/data-archive/bbob-largescale/2019/CMA_Varelas_largescale.tgz

[3]The data can be downloaded via `cocopp` or the following link: https://coco.gforge.inria.fr/data-archive/bbob-largescale/2019/sepCMA_Varelas_largescale.tgz

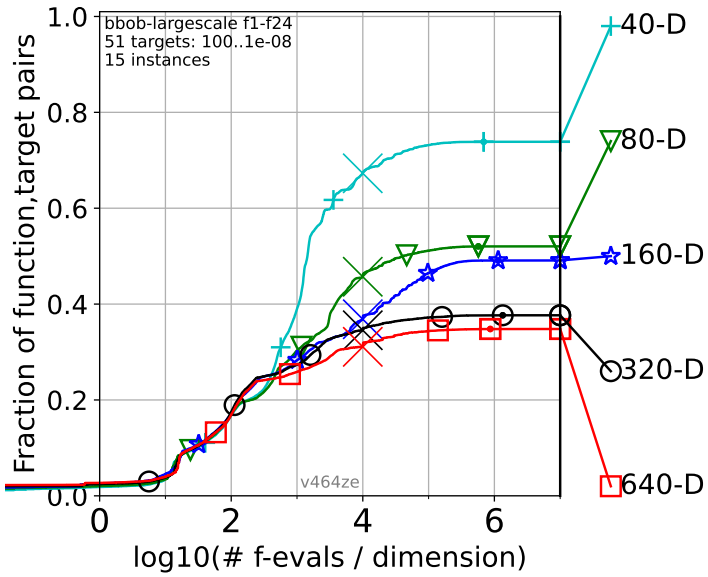
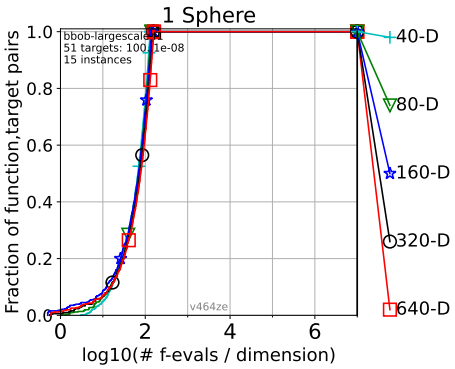


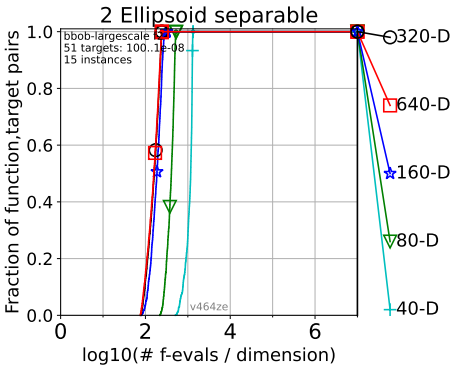
FIGURE 3.13: ECDF graph showing the performance of DiBB with CMA-ES and blocks of size 40 on the BBOB-Largescale suite for different dimensions. With increasing dimensionality, the performance gets worse. However, the performance for dimension 640 is only slightly worse than the one for dimension 320, which means that DiBB solved the problems with similar accuracy in a comparable amount of time.

multipliers. However, the difference is only small. For dimension 640, DiBB performs slightly better than sepCMA before taking the simulated restarts into account. Unfortunately, Varelas did not include data for the 640d problems for the baseline. I wanted to generate the data myself, but the large dimensionality led to slower updates of the covariance matrix, and at some point, the experiment had an estimated duration of over 20 days, which was not feasible anymore in the time frame of my thesis.

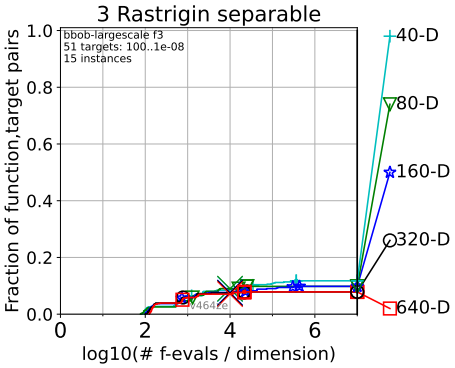
To show the differences between DiBB with CMA-ES and sepCMA on COCO, I included the ECDF comparison graphs of all 24 functions in dimension 320 in Figures 3.18 to 3.20. The plots of the 320d suite are used to point out differences between DiBB and sepCMA even if the overall performance seems to be close. Most of the curves are similar. However, sepCMA performs better than DiBB with CMA-ES and 8 blocks on the Rosenbrock function (f8; see Figure 3.18h) and the rotated Rosenbrock function (f9; see Figure 3.19a). It is likely that some of the variables in the 40 dimensional blocks did not correlate, which made it harder for DiBB to find a better solution. On the other hand, DiBB was able to find the optimal solution for the Attractive Selector (f6; see Figure 3.18f), while sepCMA converged to a solution that is much farther away. On the Step-Ellipsoidal function (f7; see Figure 3.18g), DiBB achieved the same performance as the CMA-ES baseline, while sepCMA was unable to solve the problem. Moreover, DiBB also found slightly better solutions on many other functions such as the Rastrigin (f3; see Figure 3.18c) and the Büche-Rastrigin function (f4; see Figure 3.18d), the Ellipsoidal function (f10; see Figure 3.19b), the sum of different powers (f14; see Figure 3.19f), the Schaffer functions (f18-f19; see Figures 3.20b and 3.20c), and Katsuuras (f23; see Figure 3.20g).



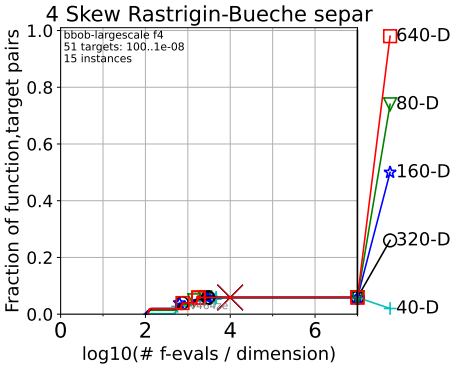
(A) Sphere function



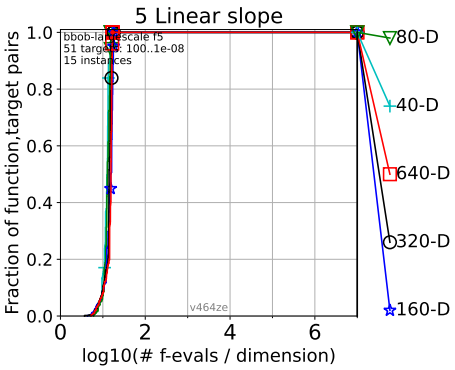
(B) Ellipsoidal function



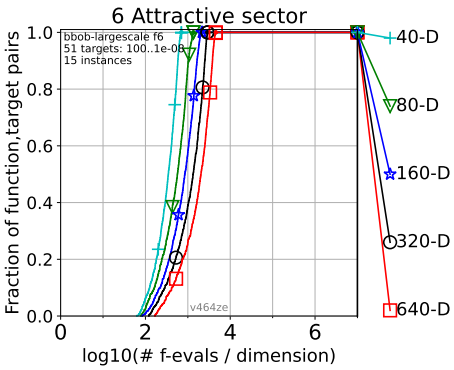
(C) Rastrigin function



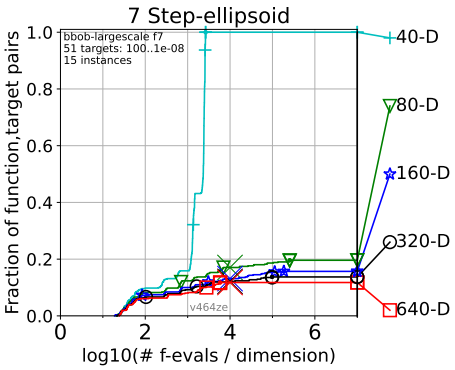
(D) Büche-Rastrigin function



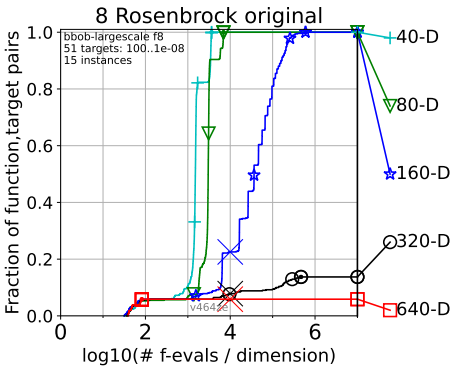
(E) Linear slope



(F) Attractive Sector function

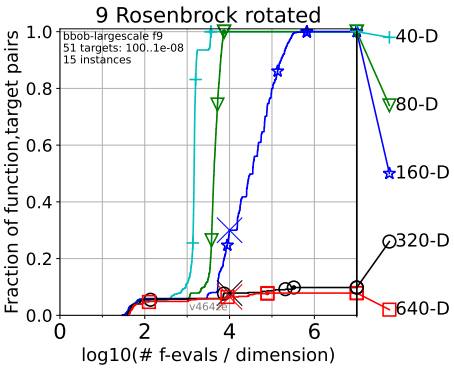


(G) Step-Ellipsoidal function

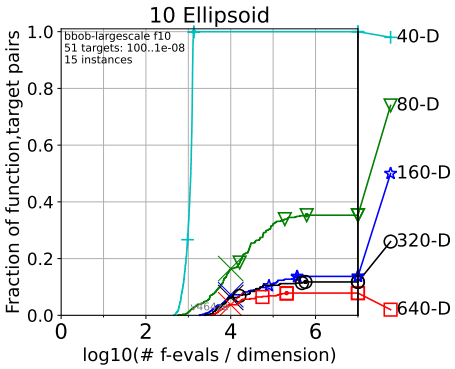


(H) Rosenbrock function, original

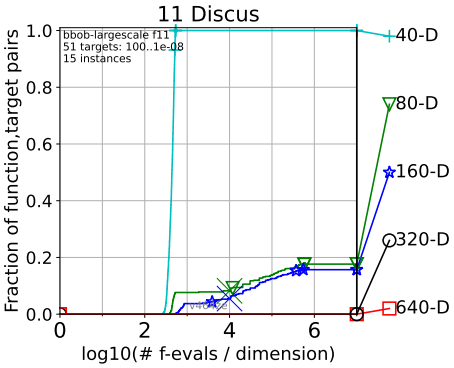
FIGURE 3.14: ECDF plots for functions f1 to f8 in the BBOB-Largescale suite, aggregated across all instances of each function. The experiment used suites of dimensions 40, 80, 160, 320, and 640 with blocks of size 40 each. Thus, the larger the dimension of a suite of problems, the more blocks were used within DiBB.



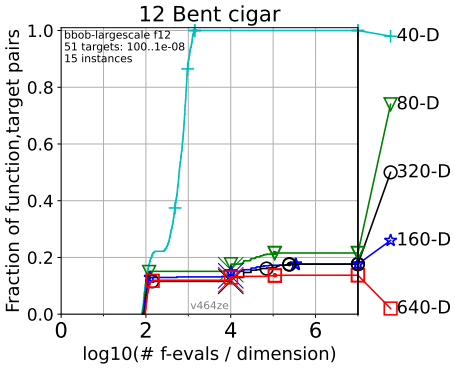
(A) Rosenbrock function, rotated



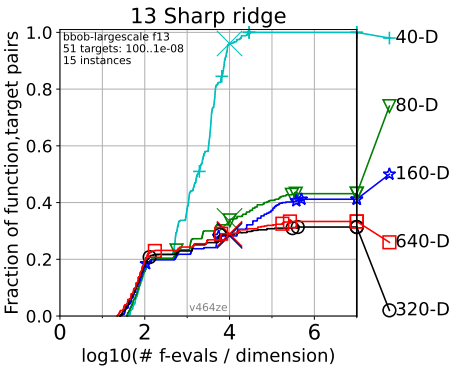
(B) Ellipsoidal function



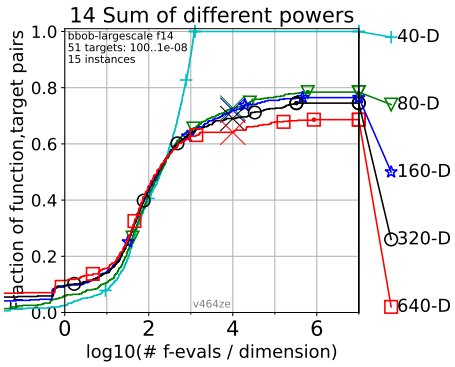
(C) Discus function



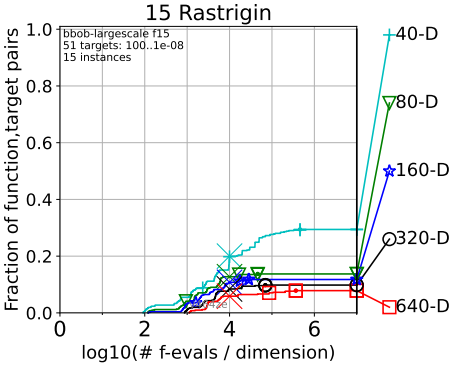
(D) Bent Cigar function



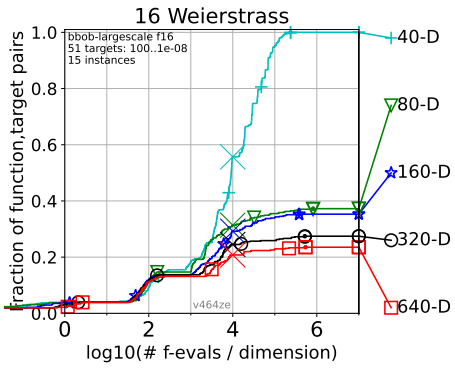
(E) Sharp Ridge function



(F) Different Powers function

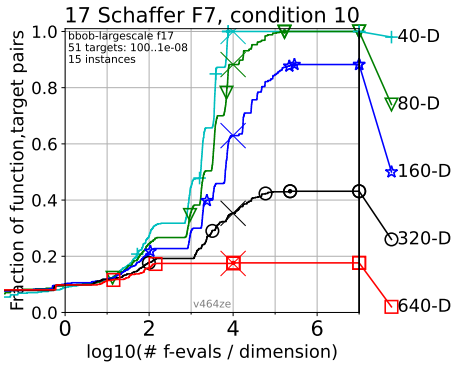


(G) Rastrigin function

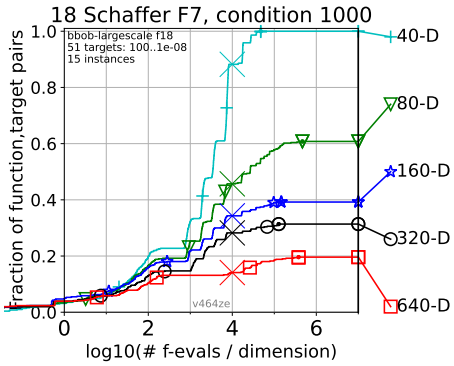


(H) Weierstrass function

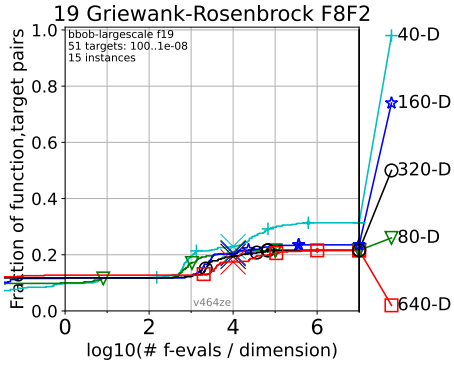
FIGURE 3.15: ECDF plots for functions f9 to f16 in the BBOB-Largescale suite, aggregated across all instances of each function. The experiment used suites of dimensions 40, 80, 160, 320, and 640 with blocks of size 40 each. Thus, the larger the dimension of a suite of problems, the more blocks were used within DiBB.



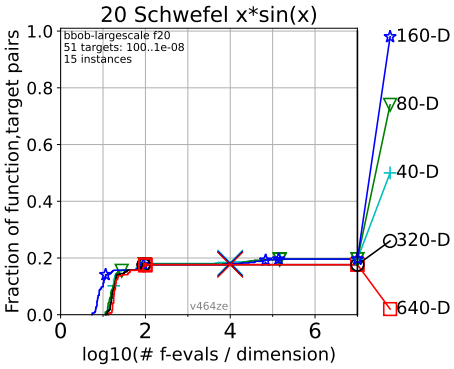
(A) Schaffers F7 function



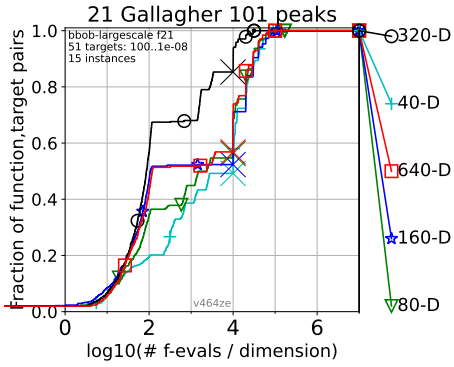
(B) Schaffers F7 function (moderately ill-conditioned)



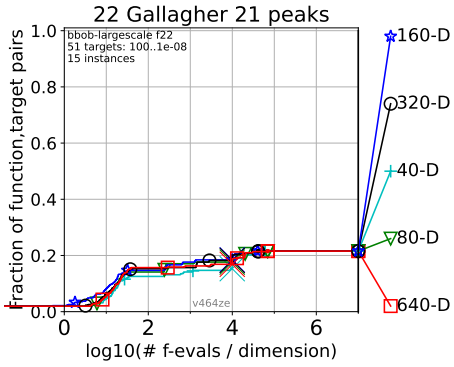
(C) Composite Griewank-Rosenbrock function F8F2



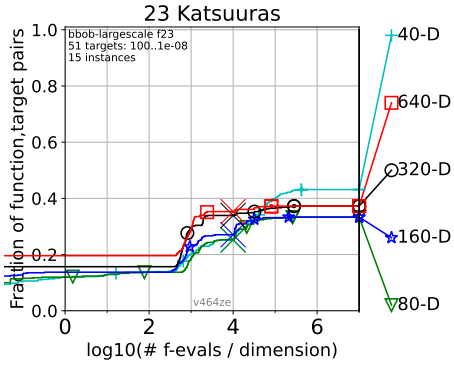
(D) Schwefel function



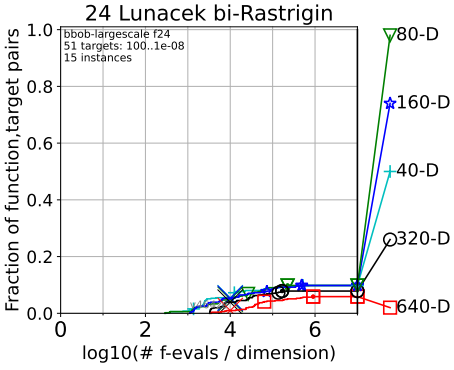
(E) Gallagher's Gaussian 101-me Peaks function



(F) Gallagher's Gaussian 21-hi Peaks function



(G) Katsuura function



(H) Lunacek bi-Rastrigin function

FIGURE 3.16: ECDF plots for functions f17 to f24 in the BBOB-Largescale suite, aggregated across all instances of each function. The experiment used suites of dimensions 40, 80, 160, 320, and 640 with blocks of size 40 each. Thus, the larger the dimension of a suite of problems, the more blocks were used within DiBB.

3.4.6 Timings

Most of the experiments were using the machines with 8 CPU cores and 32 GB RAM. However, three of the experiments used the machines with 32 CPU cores and 128 GB RAM. The three experiments were the 5d BBOB suite with one block, the 40d BBOB suite with two blocks, and the 40d BBOB-Largescale suite with one block. Because these experiments had a larger number of CPU cores at their disposal, the covariance matrix updates were likely performed faster, resulting in a decreased duration of the experiments.

Dimension	Number of blocks	Block size	Corresponding Experiments	Duration
5	1	5	`bbob_fbs`	00h 59m 41s
5	2	2, 3	`bbob_fnb`	01h 21m 05s
10	2	5	`bbob_fnb`, `bbob_fbs`	02h 05m 11s
20	4	5	`bbob_fbs`	02h 15m 40s
40	8	5	`bbob_fbs`	02h 44m 24s
20	2	10	`bbob_fnb`	04h 07m 15s
80	16	5	`bbob_ls_fnb`	04h 48m 40s
40	2	20	`bbob_fnb`	07h 32m 30s
160	16	10	`bbob_ls_fnb`	07h 49m 36s
40	1	40	`bbob_ls_fbs`	09h 48m 48s
320	16	20	`bbob_ls_fnb`	14h 02m 11s
80	2	40	`bbob_ls_fbs`	16h 08m 05s
160	4	40	`bbob_ls_fbs`	19h 20m 01s
320	8	40	`bbob_ls_fbs`	22h 38m 20s
640	16	40	`bbob_ls_fnb`, `bbob_ls_fbs`	33h 25m 22s

TABLE 3.5: Duration of the experiments in this thesis in increasing order.

Table 3.5 shows the duration of the experiments that were done in this thesis. Figure 3.21 compares the speed of DiBB with different block sizes and dimensions. It shows the duration of each of the experiments in the BBOB-Largescale suite with a fixed block size of 40. In principle, all of the machines had to do roughly the same number of evaluations since the block size was the same. However, DiBB is asynchronous and the fixed evaluation budget was not enforced on any particular machine but rather on the sum of evaluations across all machines. Moreover, the covariance matrix updates during the optimization of the 40d BBOB-Largescale suite problems were likely faster due to the increased number of CPU cores as mentioned above.

Besides that, evaluating the problems of the other dimensions should have taken approximately the same amount of time in theory. However, an overhead is introduced by using more `BlockWorkers`, since they all need to communicate. There are many reasons that could have led to this observation. For example, there could have been variations in the time that it took to evaluate the covariance matrix updates on the different blocks. Another possibility is that the `BlockWorkers` were paused more often in larger dimensions because the ran into local optima with a higher frequency.

Indeed, this is likely because increasing the dimensionality of a problem often comes with an increase in local optima.

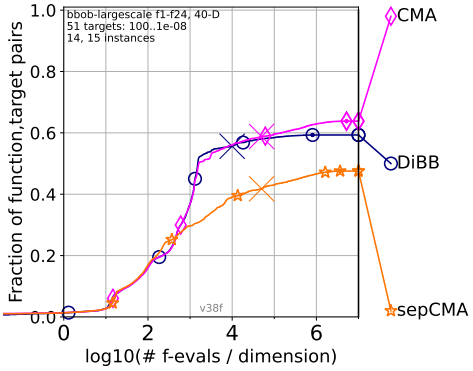
In Figure 3.22, you can see the influence of the number of blocks on the time that it takes to optimize the problems in the BBOB-Largescale suite for a given dimension and a budget multiplier of 10^4. For the 80d suite, we can see that the wall-clock time can be reduced by roughly 70% when we use 8 times more blocks. This means that we are about 3.35 times faster. If we increase the number of blocks on a 160d suite by a factor of 4, we get a decrease in wall-clock time of about 60% and we are roughly 2.47 times faster. Finally, we see that doubling the number of blocks on a 320d suite of problems leads to an experiment that runs about 1.61 times faster. Thus, the wall-clock time is reduced by roughly 38%.

Finally, I also recorded the time that it took for evaluating experiment `bbob_ls_fbs` without IPOP, which was used in Section 3.4.5 to compare the performance of DiBB to the performance of the CMA-ES baseline and sepCMA. The results can be seen in Table 3.6. We can see that the experiments took longer when the IPOP restarts were not used. This can be explained by the fact that the population sizes are smaller, which leads to more generations and thus more covariance matrix updates until reaching the evaluation budget.

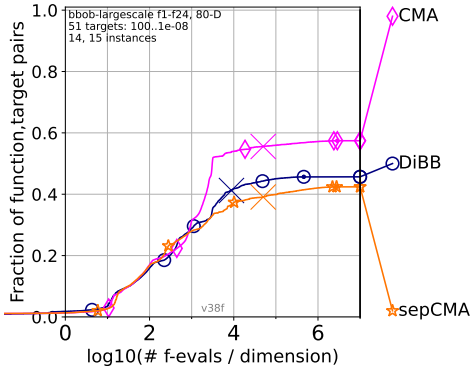
Interestingly, except the 640d suite, the wall-clock time of the experiments is much more similar than in Table 3.5. This is likely because the optimizer did not find many final targets before reaching the evaluation budget, which results in less early stops. It also shows the advantage of DiBB over sequential optimization procedures: It only took roughly 25% more time to evaluate the 320d suite in comparison to the 40d suite, even though the problem and the corresponding evaluation budget was 8 times larger.

Dimension	**Number of blocks**	**Block size**	**Duration**
40	1	40	24h 30m 16s
80	2	40	26h 13m 53s
160	4	40	27h 21m 29s
320	8	40	30h 43m 23s
640	16	40	41h 19m 55s

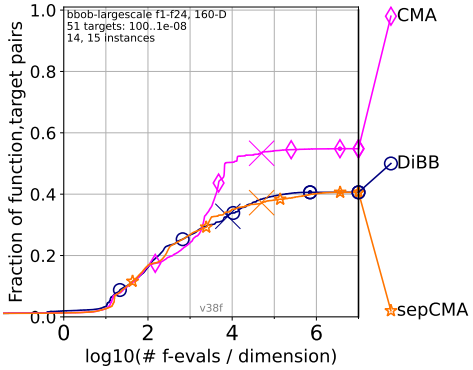
TABLE 3.6: Duration of the `bbob_ls_fbs` experiments without IPOP.



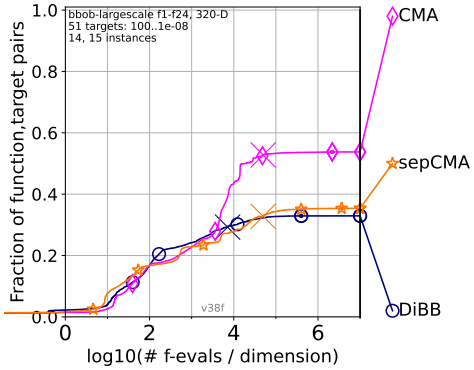
(A) Comparing the performance of DiBB with CMA-ES and one block with the performance of CMA and sepCMA on problems of dimension 40.



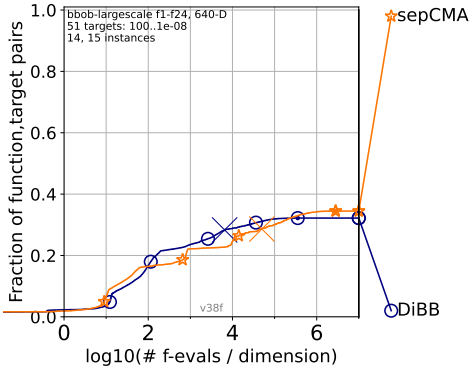
(B) Comparing the performance of DiBB with CMA-ES and two blocks with the performance of CMA and sepCMA on problems of dimension 80.



(C) Comparing the performance of DiBB with CMA-ES and four blocks with the performance of CMA and sepCMA on problems of dimension 160.

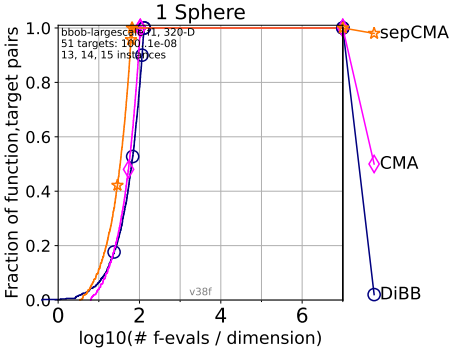


(D) Comparing the performance of DiBB with CMA-ES and eight blocks with the performance of CMA and sepCMA on problems of dimension 320.

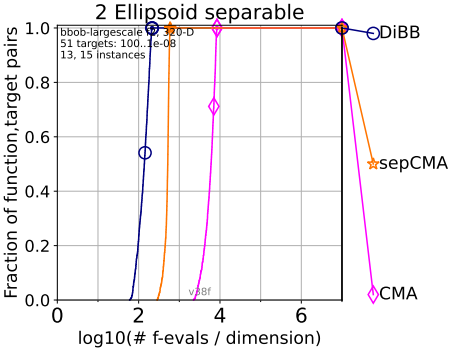


(E) Comparing the performance of DiBB with CMA-ES and sixteen blocks with the performance of CMA and sepCMA on problems of dimension 640.

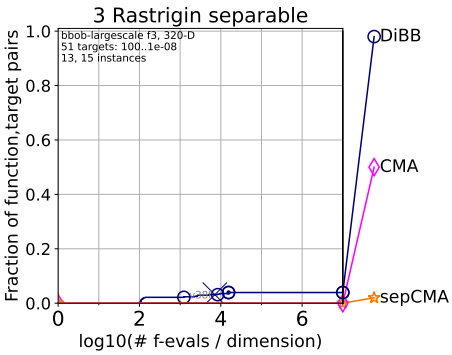
FIGURE 3.17: ECDF plots aggregated over all functions comparing the performance of DiBB with CMA-ES and blocks of size 40 with the performance of the regular CMA-ES algorithm (CMA) and a variant of it that uses a diagonal covariance matrix (sepCMA). The evaluation is done on the BBOB-Largescale suite in dimensions 40, 80, 160, 320, and 640 with a budget multiplier of 10^4. Normal restarts are used without increasing the population size (IPOP).



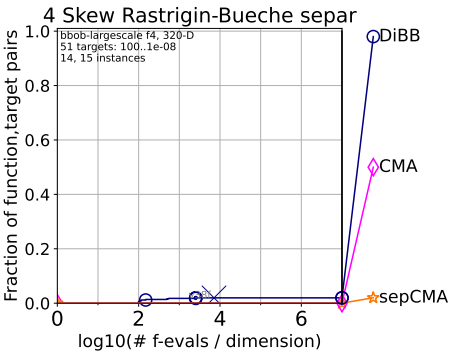
(A) Sphere function



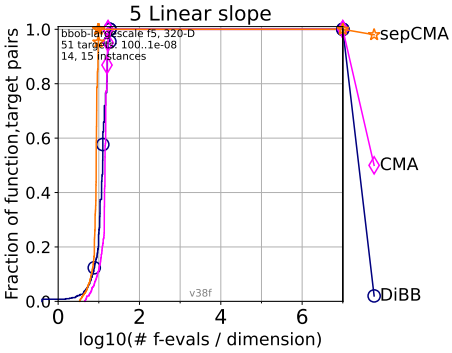
(B) Ellipsoidal function



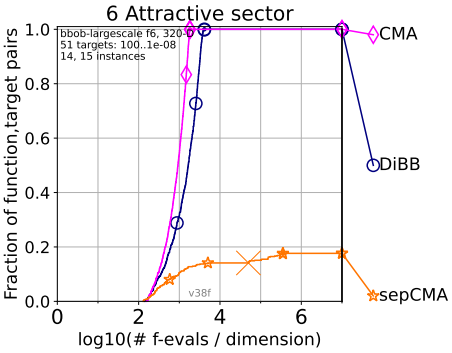
(C) Rastrigin function



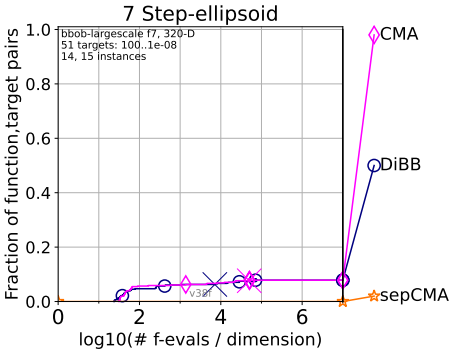
(D) Büche-Rastrigin function



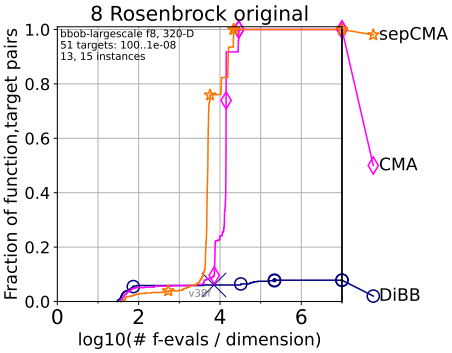
(E) Linear slope



(F) Attractive Sector function

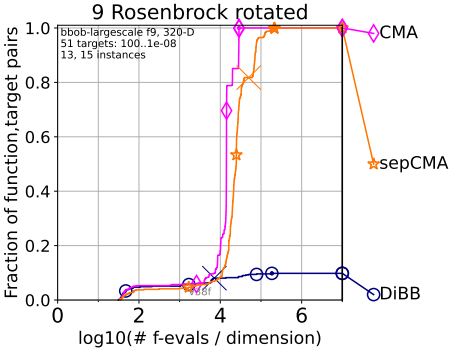


(G) Step-Ellipsoidal function

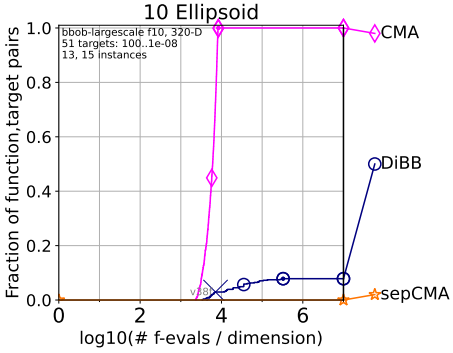


(H) Rosenbrock function, original

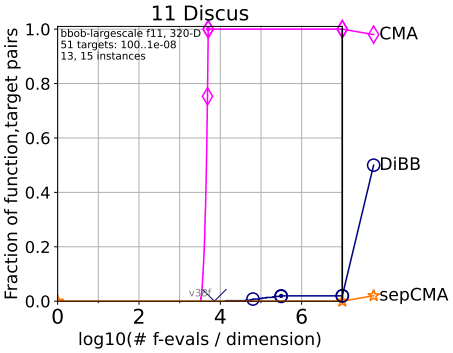
FIGURE 3.18: ECDF plots aggregated over all function instances for functions f1 to f8 in 320 dimensions for algorithms CMA-ES, DiBB with CMA-ES and 16 blocks, and sepCMA and the BBOB-Largescale suite. Normal restarts are used without increasing the population size (IPOP).



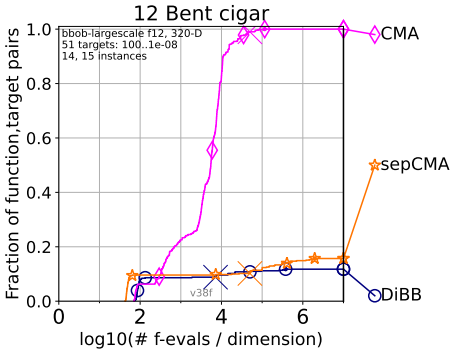
(A) Rosenbrock function, rotated



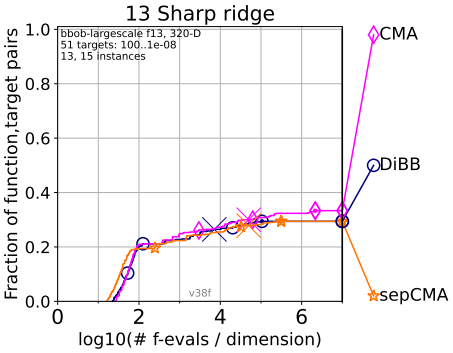
(B) Ellipsoidal function



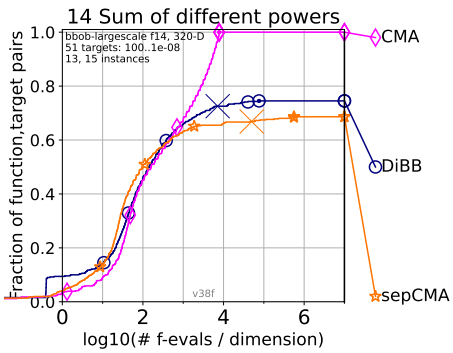
(C) Discus function



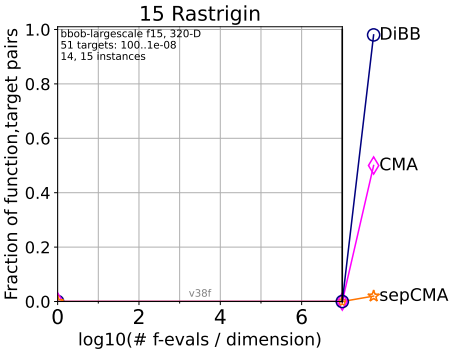
(D) Bent Cigar function



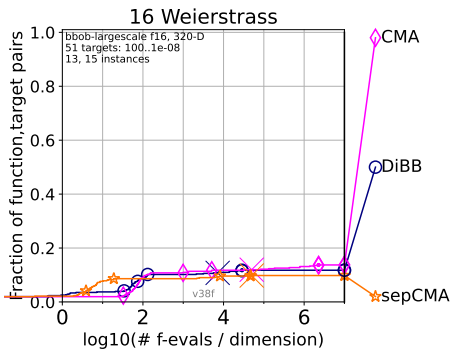
(E) Sharp Ridge function



(F) Different Powers function

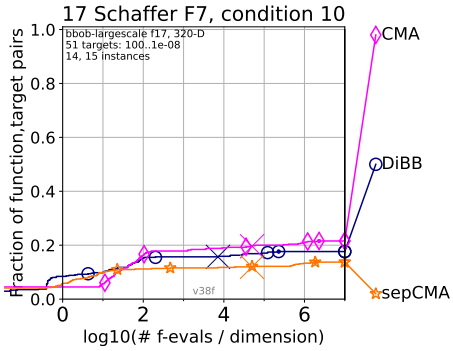


(G) Rastrigin function

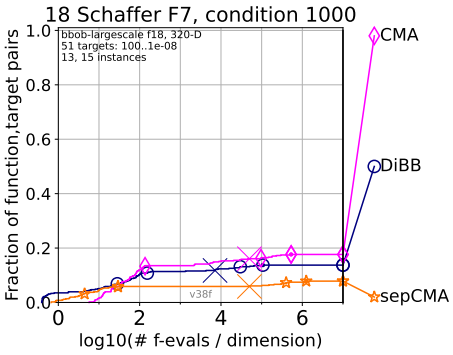


(H) Weierstrass function

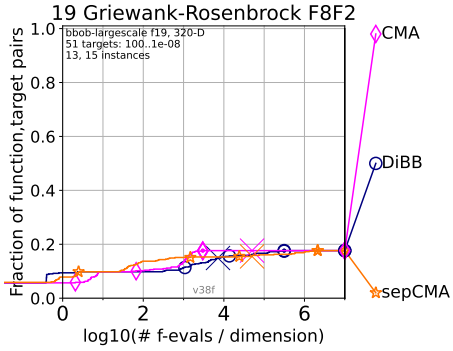
FIGURE 3.19: ECDF plots aggregated over all function instances for functions f9 to f16 in 320 dimensions for algorithms CMA-ES, DiBB with CMA-ES and 16 blocks, and sepCMA and the BBOB-Largescale suite. Normal restarts are used without increasing the population size (IPOP).



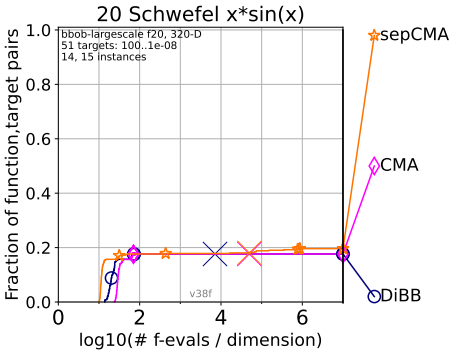
(A) Schaffers F7 function



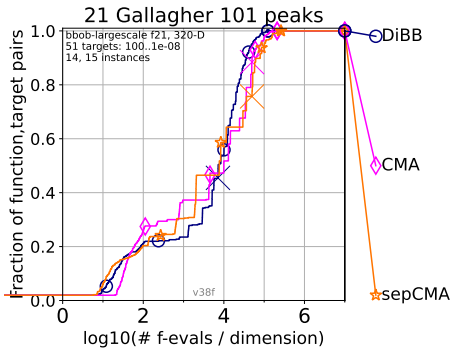
(B) Schaffers F7 function (moderately ill-conditioned)



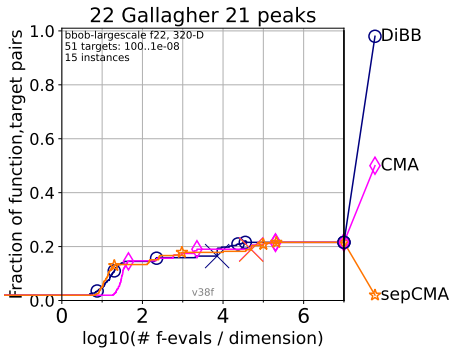
(C) Composite Griewank-Rosenbrock function F8F2



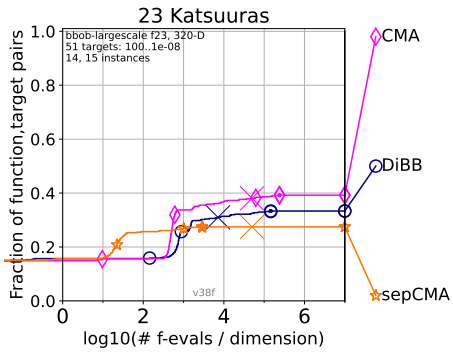
(D) Schwefel function



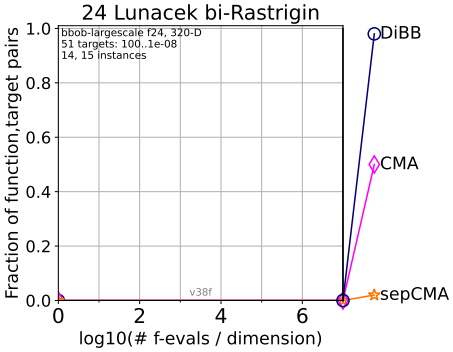
(E) Gallagher's Gaussian 101-me Peaks function



(F) Gallagher's Gaussian 21-hi Peaks function



(G) Katsuura function



(H) Lunacek bi-Rastrigin function

FIGURE 3.20: ECDF plots aggregated over all function instances for functions f17 to f24 in 320 dimensions for algorithms CMA-ES, DiBB with CMA-ES and 16 blocks, and sepCMA and the BBOB-Largescale suite. Normal restarts are used without increasing the population size (IPOP).

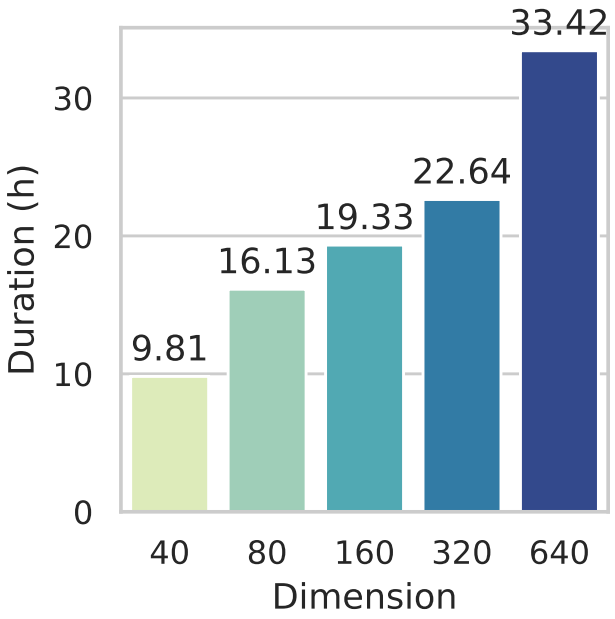
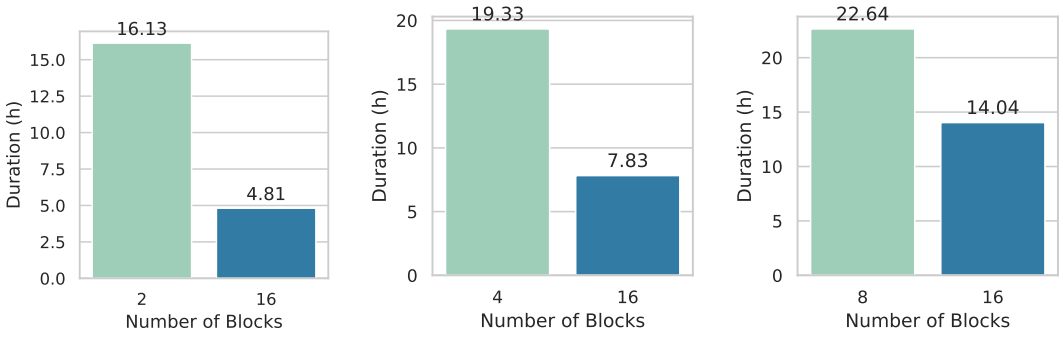


FIGURE 3.21: Durations of experiment `bbob_ls_fbs` with a fixed block size of 40 on the BBOB-Largescale. The problems were evaluated with 1, 2, 4, 8, and 16 blocks respectively and the IPOP restart strategy was used.



(A) 80d (B) 160d (C) 320d

FIGURE 3.22: Wall-clock time for the optimization of problems with dimensions 80, 160, and 320 for two different number of blocks each.

Chapter 4

Discussion and Conclusion

This thesis introduced DiBB, a novel framework for distributed black-box optimization that allows the user to divide a set of variables into blocks, which are then optimized on different machines. With this method, larger problems can be tackled and because the different blocks are optimized in parallel, the wall-clock time of the optimization procedure is significantly decreased. To investigate these assumptions and the practicality of DiBB, three research questions were presented in Section 3.1. I will now answer these three questions using the results that were reported in Section 3.4.

My thesis shows that using DiBB is a promising path for tackling larger problems with black box optimization that were not feasible until now. Instead of using no correlational information at all, as it has been done by previous methods, DiBB leverages the tradeoff between speed (wall-clock time) and accuracy (sample efficiency). Most of the time, this novel method should outperform the previous approaches with the diagonal covariance matrix.

RQ1: How does DiBB compare to optimization algorithms that use the entire correlational information? The results show that DiBB with CMA-ES can indeed produce comparable results to a CMA-ES algorithm that uses the entire covariance matrix. However, this depends on the number of blocks. The more blocks that we use, the larger the gap in performance between CMA-ES and DiBB with CMA-ES. This leads to the heuristic that DiBB should always make use of the largest block size that is still feasible given a certain budget of time. In the BBOB-Largescale experiment with a fixed number of blocks, we have seen that CMA-ES found a better solution for the Gallagher 101 peaks problem in 320 dimensions and blocks of size 20 than in all other dimensions with different block sizes. This shows that the heuristic that was mentioned before is not always true. Indeed, the best scenario is the one in which we can deduce the blocks of parameters that correlate or in which we at least have justifiable assumptions. This is of particular relevance for applications in Neuroevolution, where the correlations can be inferred from the network structure.

In Section 3.4.4 we have seen that there can be problems where CMA-ES does not benefit from the full covariance matrix. Generally speaking, CMA-ES benefits the most from a full covariance matrix when all of the parameters are correlated. If a subset of them is uncorrelated, CMA-ES has to learn this information first, which takes some time. Additionally, even if CMA-ES learned that some of the parameters do not correlate, the dimension of the covariance matrix will still be the same, and all of the covariance matrix updates will still be slower than if we were using multiple blocks of correlated parameters.

Moreover, we have seen in Section 3.4.5 that DiBB performs better than the separable version of CMA-ES with a diagonal covariance matrix on many problems. This means that DiBB successfully leveraged the trade-off between a full covariance matrix and no correlational information at all. In addition, both the standard CMA-ES

algorithm as well as the variant with a diagonal covariance matrix can be generalized with DiBB by selecting the corresponding number of blocks.

RQ2: How much faster can DiBB run an experiment with a fixed budget of function evaluations? In Section 3.4.6 we have seen that DiBB completed runs on a given dimensionality faster the more blocks that were used. We found out that there are some overheads which result in an increased wall-clock time of the experiment for larger dimensions, even if using the same number of evaluations per machine. I pointed out that the pausing mechanism of the `BlockWorkers` was most likely the main reason behind this observation. It is important to note that we are talking about fixed evaluation budgets. On the other hand, if we looked at the time that it took the optimizer to find a solution smaller than a specified fitness score, the wall-clock time of DiBB could in principle be the same or maybe also worse than the one of CMA-ES without DiBB.

This has to do with the fact that DiBB has a smaller sample efficiency, i.e. that the generations within DiBB happen faster, but they do not have the same amount of information as CMA-ES that uses all possible correlations. However, this is highly dependent on the number of blocks and their sizes. Furthermore, if we run both CMA-ES and DiBB with CMA-ES for the same amount of time, DiBB will evaluate significantly more individuals because they are evaluated in parallel. We have seen that we were about 1.61 times faster when we used two blocks rather than one on the 80d BBOB-Largescale suite. If we were to rerun DiBB with a fixed wall-clock time budget rather than focusing on the number of evaluations, DiBB would perform also about 1.61 times more evaluations. In principle, this should lead to an increased performance when using a fixed budget of time. On some problems (particularly if separable) it may even be possible to outperform the standard CMA-ES algorithm with DiBB and CMA-ES just because it can do more evaluations in less time.

RQ3: Does DiBB's performance scale to high-dimensional problems? I was unable to run the standard CMA-ES algorithm on the 640d BBOB-Largescale suite with a budget multiplier of 10^4 due to time limitations. As mentioned earlier, it would have taken more than 20 days, which was unfeasible in my situation. However, I was able to run the suite with DiBB and CMA-ES in 33.5 hours. Often, a problem could theoretically be optimized by CMA-ES given sufficient resources, but it would take too long to do so. Therefore, time is one of the most limiting factors of CMA-ES, which decides if a problem is feasible or not.

If the estimation of 20 days was correct for the CMA-ES algorithm, then we increased the speed of the experiment at least by a **factor of 14** using DiBB with CMA-ES and 16 blocks. Given a fixed budget of time and an unlimited number of machines, we could essentially make any problem dimensionality feasible with DiBB. In practice, this is not entirely true because on non-separable problems, which are common, DiBB performs worse with an increasing number of blocks. Therefore, the answer is somewhere in between. For example, if we know the largest possible problem for the standard CMA-ES algorithm given a fixed budget of time, we are able to double the dimension of the problem and solve it with DiBB and CMA-ES using 2 blocks. Because there is often a overhead, we might want to increase the number of blocks to make it actually feasible within the time budget.

This leads to the conclusion that we can tackle much larger problems if we use more than one block of parameters within DiBB. How much the performance of an

optimization algorithm suffers when we divide the optimization variables into different blocks depends on the problem and the optimizer that is being used. Therefore, this is an open subject of investigation.

4.1 Future Work

There is much room for further research regarding DiBB. For example, more data needs to be collected about the performance of different optimization algorithms with DiBB. Another important direction is the application of DiBB to other benchmarks and problems. This is especially true for problems that were infeasible so far. An important area of research where DiBB should be evaluated is Neuroevolution in general. Especially in Direct Policy Search on RL problems, DiBB could potentially lead to results competitive with deep learning.

Another important topic of research is how to decide the number of blocks or the size of the blocks that DiBB should use on different classes of problems. For example, heuristics should be investigated on how to choose the blocks if we want to optimize a neural network with DiBB. These heuristics will be dependent on the network type, so there have to be different studies for Multi-Layer Perceptrons, Convolutional Neural Networks, Recurrent Neural Networks, etc.

Moreover, we have seen that even though CMA-ES may learn that some of the parameters are uncorrelated, the size of the covariance matrix will still be the same, which results in the same time complexity. The same can happen to DiBB if the blocks are chosen poorly. An interesting approach would be to extend and study DiBB with a mechanism that automatically splits a block into two or more blocks once it learns that one or multiple parameters in the original block are uncorrelated. This should further increase the wall-clock speed.

The important part about DiBB is not the implementation but the concept behind it. Further investigations has to be done on how to improve the framework such that it creates as little overhead as possible. Modifications may include switching from Ray to another library for distributed computation (e.g. ZeroMQ) or porting DiBB to another programming language.

Bibliography

Auger, Anne and Nikolaus Hansen (2005). "A restart CMA evolution strategy with increasing population size". In: *2005 IEEE Congress on Evolutionary Computation*. Vol. 2, 1769–1776 Vol. 2. DOI: 10.1109/CEC.2005.1554902.

Bunge, Mario (1963). "A General Black Box Theory". In: *Philosophy of Science* 30.4. _eprint: https://doi.org/10.1086/287954, pp. 346–358. DOI: 10.1086/287954. URL: https://doi.org/10.1086/287954.

Conti, Edoardo et al. (2018). "Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pp. 5032–5043.

Cuccu, Giuseppe (2018). "Extending the Applicability of Neuroevolution". PhD Thesis. Fribourg, Switzerland: Université de Fribourg. URL: https://exascale.info/assets/pdf/cuccu2018phd.pdf.

Elhara, Ouassim Ait et al. (2019). "COCO: The Large Scale Black-Box Optimization Benchmarking (bbob-largescale) Test Suite". In: *arXiv preprint arXiv:1903.06396*.

Glasmachers, Tobias and Oswin Krause (2020). "The Hessian Estimation Evolution Strategy". In: *International Conference on Parallel Problem Solving from Nature*. Springer, pp. 597–609.

Glasmachers, Tobias, Tom Schaul, et al. (2010). "Exponential Natural Evolution Strategies". In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. Issue: CONF, pp. 393–400.

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

— (2005). "The CMA Evolution Strategy: A Tutorial". In:

— (July 2019). "A global surrogate assisted CMA-ES". en. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Prague Czech Republic: ACM, pp. 664–672. ISBN: 978-1-4503-6111-8. DOI: 10.1145/3321707.3321842. URL: https://dl.acm.org/doi/10.1145/3321707.3321842.

Hansen, Nikolaus, Youhei Akimoto, and Petr Baudis (Feb. 2019). *CMA-ES/pycma on Github*. Published: Zenodo, DOI:10.5281/zenodo.2559634. DOI: 10.5281/zenodo.2559634. URL: https://doi.org/10.5281/zenodo.2559634.

Hansen, Nikolaus, Anne Auger, Steffen Finck, et al. (2010). *Real-Parameter Black-Box Optimization Benchmarking 2010: Experimental Setup*. Research Report RR-7215. INRIA.

Hansen, Nikolaus, Anne Auger, Raymond Ros, et al. (Jan. 2021). "COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting". In: *Optimization Methods and Software* 36.1. arXiv: 1603.08785, pp. 114–144. ISSN: 1055-6788, 1029-4937. DOI: 10.1080/10556788.2020.1808977. URL: http://arxiv.org/abs/1603.08785 (visited on 06/21/2021).

Hansen, Nikolaus, Dimo Brockhoff, et al. (Mar. 2019). *COmparing Continuous Optimizers: numbbo/COCO on Github*. Version v2.3. DOI: 10.5281/zenodo.2594848. URL: https://doi.org/10.5281/zenodo.2594848.

Hansen, Nikolaus, Steffen Finck, et al. (2009). "Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions". In: Publisher: Citeseer.

Hansen, Nikolaus and Stefan Kern (2004). "Evaluating the CMA Evolution Strategy on Multimodal Test Functions". In: *Parallel Problem Solving from Nature-PPSN VIII: 8th International Conference, Birmingham, UK, September 18-22, 2004, Proceedings*. Vol. 8. Springer Science & Business Media, p. 282.

Krause, Oswin, Dídac Rodríguez Arbonès, and Christian Igel (2016). "CMA-ES with Optimal Covariance Update and Storage Complexity". In: *Advances in Neural Information Processing Systems* 29, pp. 370–378.

Lehman, Joel (2012). "Evolution Through The Search For Novelty". PhD Thesis. University of Central Florida Orlando, Florida.

Mousavirad, Seyed Jalaleddin, Azam Asilian Bidgoli, and Shahryar Rahnamayan (Aug. 2019). "Tackling Deceptive Optimization Problems Using Opposition-based DE with Center-based Latin Hypercube Initialization". In: *2019 14th International Conference on Computer Science Education (ICCSE)*. ISSN: 2473-9464, pp. 394–400. DOI: 10.1109/ICCSE.2019.8845360.

Rastrigin, Leonard Andreevi (1974). "Extremal Control Systems". Russian. In: *Theoretical Foundations of Engineering Cybernetics Series* 3. Publisher: Nauka.

Rechenberg, Ingo (1989). "Evolution Strategy: Nature's Way of Optimization". In: *Optimization: Methods and Applications, Possibilities and Limitations*. Springer, pp. 106–126.

Ros, Raymond and Nikolaus Hansen (2008). "A Simple Modification in CMA-ES Achieving Linear Time and Space Complexity". In: *Parallel Problem Solving from Nature – PPSN X*. Ed. by Günter Rudolph et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 296–305. ISBN: 978-3-540-87700-4.

Rudolph, Günter (1990). "Globale Optimierung mit parallelen Evolutionsstrategien". German. PhD Thesis. Diplomarbeit, Universit at Dortmund, Fachbereich Informatik.

Sauppe, J., ed. (2006–18). *Mathematical Programming Glossary*. Originally authored by Harvey J. Greenberg, 1999-2006. http://glossary.computing.society.informs.org: INFORMS Computing Society.

Schaul, Tom, Tobias Glasmachers, and Jürgen Schmidhuber (2011). "High Dimensions and Heavy Tails for Natural Evolution Strategies". In: *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 845–852.

Varelas, Konstantinos (2019). "Benchmarking Large Scale Variants of CMA-ES and L-BFGS-B on the Bbob-Largescale Testbed". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. event-place: Prague, Czech Republic. New York, NY, USA: Association for Computing Machinery, pp. 1937–1945. ISBN: 978-1-4503-6748-6. DOI: 10.1145/3319619.3326893. URL: https://doi.org/10.1145/3319619.3326893.