Benchmarking a Pool-Based Execution with GA and PSO Workers on the BBOB Noiseless Testbed

Mario García-Valdez
Instituto Tecnolgico de Tijuana
Tijuana BC, Mexico
mario@tectijuana.edu.mx

JJ Merelo
Grupo GeNeura, Depto. ATC + CITIC, Universidad de Granada
Granada, Spain
jmerelo@ugr.es

ABSTRACT

In this work, we evaluate an asynchronous population-based algorithm following a pool-based approach. In Pool-based algorithms, a collection of workers collaborates through a shared population repository. In particular, we followed the EvoSpace approach in which workers asynchronously interact with a population pool by taking samples of the population to perform a standard search on the samples, to then return newly evolved solutions back to the pool. For this purpose, we use the BBOB Noiseless Testbed and a hybrid algorithm combining two kinds or workers: PSO and GA. We find that a Pool- based approach outperforms the canonical GA and PSO algorithms in nearly all cases. The results of these tests suggest that a Pool Based approach can be used to implement hybrid algorithms that can improve the performance of canonical population-based optimization algorithms.

CCS CONCEPTS

•**Computing methodologies → Continuous space search;**

KEYWORDS

Benchmarking, Black-box optimization, Nature-inspired metaheuristics, Distributed Evolutionary Algorithms

ACM Reference format:
Mario García-Valdez and JJ Merelo. 2017. Benchmarking a Pool-Based Execution with GA and PSO Workers on the BBOB Noiseless Testbed. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3067695.3086573

1 INTRODUCTION

Asynchronous EAs [1, 2, 14] have started to become common only relatively recently, in an effort to exploit computing resources available through different Internet technologies, including cloud and volunteer-based. In this work, we are interested in benchmarking asynchronous EAs following a pool-based approach, we will refer to such algorithms as *Pool-based EAs* or PEAs, and highlight the fact that such systems are intrinsically parallel, distributed and asynchronous.

Pool-EAs differ from the closely related Island Model, mainly with regards to the responsibilities assigned to the server. When there is a server in the island model, it is responsible for the interaction and synchronization of all the populations. In Pool-EAs, on the other hand, the population repository only receives stateless requests from isolated workers or clients. In this way, Pool-EAs are capable of using and leveraging an ad-hoc and ephemeral collaboration of computing resources.

The algorithm presented in this paper follows the basic EvoSpace model [7] in which EvoWorkers asynchronously interact with the population pool by taking samples of the population to perform a standard evolutionary search on the samples, to then return newly evolved solutions back to the pool. This design is a particular instance of a Pool-based EA, which, as long as there is a shared population pool leaves every other detail to their specific implementations.

To test this PEA against the Noiseless BBOB testbed, we mixed two canonical versions of the GA and PSO algorithms, by just defining EvoWorkers of each kind. The performance obtained highlight that a Pool Based approach can be used to improve the performance of single population-based optimization algorithms.

2 ALGORITHM PRESENTATION

As we mentioned earlier, EvoWorkers are independent of the population repository, and developers can code them in any language that supports HTTP requests. In this work, EvoWorkers were implemented in Python taking advantage of two open source libraries of nature inspired optimization metaheuristics: DEAP [6] and EvoloPy [4]. The EvoSpace population repository is implemented in Node.js and uses the Redis memory store as the repository. The code is in the following GitHub repository: https://github.com/mariosky/evospace-js. On the other hand, each EvoWorker runs inside a Docker container. Before each experiment, a script initializes the population on the server, creating the number of individuals specified in the *Population Size* parameter, this size depends on the dimension of the problem. The same script then starts the number of containers needed, one for each EvoWorker. When starting each container, the following parameters are used: first, the *Sample Size* indicates the number of individuals the worker would take from the server on each interaction, then the *Iterations per Sample* parameter specifies the number of generations or iterations the local algorithm will run before sending back to the server the resulting population. Finally, the number of times an EvoWorker will take, evolve and return a sample, is indicated by the *Samples per Worker* parameter.

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3086573

Table 1: EvoWorker Setup

Dimension	2	3	5	10	20	40
Iterations per Sample	50	50	50	50	50	50
Sample Size	100	100	100	200	200	200
Samples per Worker	20	30	25	25	25	25
PSO Workers	1	1	2	2	4	8
GA Workers	1	1	2	2	4	8

Table 2: DEAP GA EvoWorker Parameters

Selection	Tournament size=12
Mutation	Gaussian $\mu = 0.0$, $\sigma = 0.5$, indbp=0.05
Mutation Probability	[.1,.6]
Crossover	Two Point
Crossover Probability	[.8,1]

Table 3: EvoloPy PSO EvoWorker Parameters

V_{max}	6
W_{max}	0.9
W_{min}	0.2
C_1	2
C_2	2

These parameters are specified for each of the dimensions, and they give the maximum number of function evaluations for each problem. Each time an EvoWorker returns an evolved sample, it also returns benchmark data, indicating the type of algorithm and parameters used, the number of function evaluations, best individual, and fitness of each iteration. The source code for the Python EvoWorkers proposed in this work is in the following GitHub repository: https://github.com/mariosky/EvoWorker. For the purpose of this work, we called this algorithm EvoSpace-PSO-GA.

3 EXPERIMENTAL PROCEDURE

A script was responsible for creating the EvoWorker containers and running the testbed. The maximum number of function evaluations was set to $10^5 * D$. For each run of the algorithms, the initial solution X_0 is sampled uniformly in [-5, 5]^D. In order to maintain the required number of function evaluations the EvoWorkers were set up as shown in Table 1. The current experiment uses instances 1-5 and 41-50 from the 2016 version of the test bed. In this experiment the maximum dimensionality was 20.

3.1 Parameter Tuning

The parameters for each type of EvoWorker are shown in Table 2 and Table 3 for GAs and PSO respectively. These parameters were obtained by testing first on the Rastrigin separable function with five dimensions. After about fifteen experiments the most challenging targets were achieved. We tested again with functions one to three, and after obtaining favorable results, the PSO and GA algorithm parameters were set.

4 CPU TIMING

In order to evaluate the CPU timing of the algorithm, we have run the EvoSpace-PSO-GA algorithm on the BBOB test suite [11] with no restarts for a maximum budget equal to $10^5 * dim$ function evaluations according to [13]. The Python (version 2.7.13) code was run on a MacPro (late 2013) Intel(R) Quad Core Intel Xeon (TM) E5 CPU @ 3.7GHz with 12 GB 1866 MHz DDR3 ECC RAM, 1 processor and 4 cores, EvoWorkers were executed in Docker (v 17.03.0-ce-mac2) container images are available at https://hub.docker.com/r/mariosky/evo_worker/ . The time per function evaluation for dimensions 2, 3, 5, 10, 20, 40 equals 0.0678, 0.0676, 0.0372, 0.03929, 0.03929, and 0.05223 milliseconds respectively. Times were measured with EvoWorkers executing in parallel on the Docker containers mentioned above, is important to notice that the number of EvoWorkers increased with dimensions (see Table 1).

5 RESULTS

After the execution, a script processed the logs and generated the files needed by the COCO platform [10] post-processing scripts.

A requirement of the COCO platform is that it needs to inspect each function evaluation to keep the log required to analyze the execution. The logging code maintains a sequential record of the number function calls.

It is not practical to track the exact sequential number of function evaluations in an asynchronous execution, because many workers could be calling the function at the same time. For this reason, the granularity of the number of function evaluations and their order was kept at the sample and iteration level. As we mentioned earlier, each worker returns the number of evaluations performed in each iteration. The order of function calls was given by the order in which the server received the samples and the order of the iterations in each. On the other hand, the number of function evaluations is incremented in each iteration by the sample size and the best function evaluation is assigned that number, as if in each iteration the best solution was found in the last function evaluation. Instead of increasing the number by one it is incremented by the number of solutions in the sample. It is important to notice that EvoWorkers run the algorithm only for a small number of iterations and with a relatively small sample of the population. For instance, for the COCO benchmark presented in the case study the maximum number of function evaluations in a single iteration (generation) was 200.

The COCO post-processing script currently has an assertion stating that all the function evaluations start at number 1. In our case the first evaluation had a number equal to the total number of evaluations in a single iteration. In order to run the script a function evaluation of a random generated solution was inserted at the beginning. Each component of the solution was generated using the basic random function from the Python standard library. The random function generates a random float uniformly in the semi-open range [0.0, 1.0) using the Mersenne Twister as the core generator.

Results of EvoSpace from experiments according to [13] and [9] on the benchmark functions given in [5, 12] are presented in Figures 1 and 2 and in Table 3. The experiments were performed

with COCO [10], version bbob.v15.03 in python, the plots were produced with version 2.0.

6 DISCUSSION

The results obtained show that the algorithm reaches the most difficult targets on separable functions (1-5), and scales well to higher dimensions, with better-than-linear scaling in most cases and close to quadratic in some of them. These results are competitive when compared to other nature-inspired algorithms [8]. In particular when comparing against the Binary GA algorithm implementation of Nicolau [16], and the PSO algorithm by El-Abd and Kamel [3]: the EvoSpace-PSO-GA algorithms outperforms both, reaching more targets with a lower expected running time (ERT). When testing with harder problems, for instance functions 13 and 24, the algorithm is not capable of reaching the most difficult targets.

Future lines of work will focus on using other EA or metaheuristic techniques, such as the Grey Wolf Optimizer [15] or Differential Evolution [17] for having workers that are heterogeneous in more than one sense. RPSS could be used in those cases where each algorithm has a different set of parameters, but also to randomly select the technique employed.

The experiment shows that an asynchronous execution of population-based optimization algorithms following a Pool-based approach is possible and easy to achieve. Results, however, are still preliminary and further tuning of the parameters could potentially yield better results.

ACKNOWLEDGMENTS

This work has been supported in part by Conacyt Project PROINNOVA-220590 and Ministerio español de Economía y Competitividad under project TIN2014-56494-C4-3-P (UGR-EPHEMECH).

REFERENCES

[1] Enrique Alba and José M Troya. 2001. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems* 17, 4 (2001), 451–465.

[2] J. Atienza, P. A. Castillo, M. García, J. González, and J.J. Merelo. 2000. Jenetic: a distributed, fine-grained, asynchronous evolutionary algorithm using Jini. In *Proc. JCIS 2000 (Joint Conference on Information Sciences)*, P. P. Wang (Ed.), Vol. I. 1087–1089. ISBN: 0-9643456-9-2.

[3] Mohammed El-Abd and Mohamed S Kamel. 2009. Black-box optimization benchmarking for noiseless function testbed using particle swarm optimization. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*. ACM, 2269–2274.

[4] Hossam Faris, Ibrahim Aljarah, Seyedali Mirjalili, Pedro A. Castillo, and Juan J. Merelo. 2016. EvoloPy: An Open-source Nature-inspired Optimization Framework in Python. In *Proceedings of the 8th International Joint Conference on Computational Intelligence - Volume 1: ECTA, (IJCCI 2016)*. 171–177. DOI: http://dx.doi.org/10.5220/0006048201710177

[5] S. Finck, N. Hansen, R. Ros, and A. Auger. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Presentation of the Noiseless Functions.* Technical Report 2009/20. Research Center PPE. http://coco.lri.fr/downloads/download15.03/bbobdocfunctions.pdf Updated February 2010.

[6] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research* 13, Jul (2012), 2171–2175.

[7] Mario García-Valdez, Leonardo Trujillo, Juan-J Merelo, Francisco Fernández de Vega, and Gustavo Olague. 2015. The EvoSpace Model for Pool-Based Evolutionary Algorithms. *Journal of Grid Computing* 13, 3 (2015), 329–349. DOI: http://dx.doi.org/10.1007/s10723-014-9319-2

[8] Anne Augerfi!?Steffen Finckfi!?Nikolaus Hansen and Raymond Ros. 2010. BBOB 2009: Comparison Tables of All Algorithms on All Noiseless Functions. (2010).

[9] N. Hansen, A Auger, D. Brockhoff, D. Tušar, and T. Tušar. 2016. COCO: Performance Assessment. (2016).

[10] N. Hansen, A. Auger, O. Mersmann, T. Tušar, and D. Brockhoff. 2016. COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting. (2016). arXiv:1603.08785.

[11] N. Hansen, S. Finck, R. Ros, and A. Auger. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions.* Technical Report RR-6829. INRIA. http://hal.inria.fr/inria-00362633/en/

[12] N. Hansen, S. Finck, R. Ros, and A. Auger. 2009. *Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions.* Technical Report RR-6829. INRIA. http://coco.gforge.inria.fr/bbob2012-downloads Updated February 2010.

[13] N. Hansen, T. Tušar, O. Mersmann, A. Auger, and D. Brockhoff. 2016. COCO: The Experimental Procedure. (2016). arXiv:1603.08776.

[14] Juan J. Merelo, Antonio M. Mora, Pedro A. Castillo, Juan L. J. Laredo, Lourdes Araujo, Ken C. Sharman, Anna I. Esparcia-Alczar, Eva Alfaro-Cid, and Carlos Cotta. 2008. Testing the Intermediate Disturbance Hypothesis: Effect of Asynchronous Population Incorporation on Multi-Deme Evolutionary Algorithms. In *Parallel Problem Solving from Nature - PPSN X (LNCS)*, Gunter Rudolph, Thomas Jansen, Simon Lucas, Carlo Poloni, and Nicola Beume (Eds.), Vol. 5199. Springer, Dortmund, 266–275. DOI: http://dx.doi.org/10.1007/978-3-540-87700-4_27

[15] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. 2014. Grey wolf optimizer. *Advances in Engineering Software* 69 (2014), 46–61.

[16] Miguel Nicolau. 2009. Application of a simple binary genetic algorithm to a noiseless testbed benchmark. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*. ACM, 2473–2478.

[17] Rainer Storn and Kenneth Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization* 11, 4 (1997), 341–359.

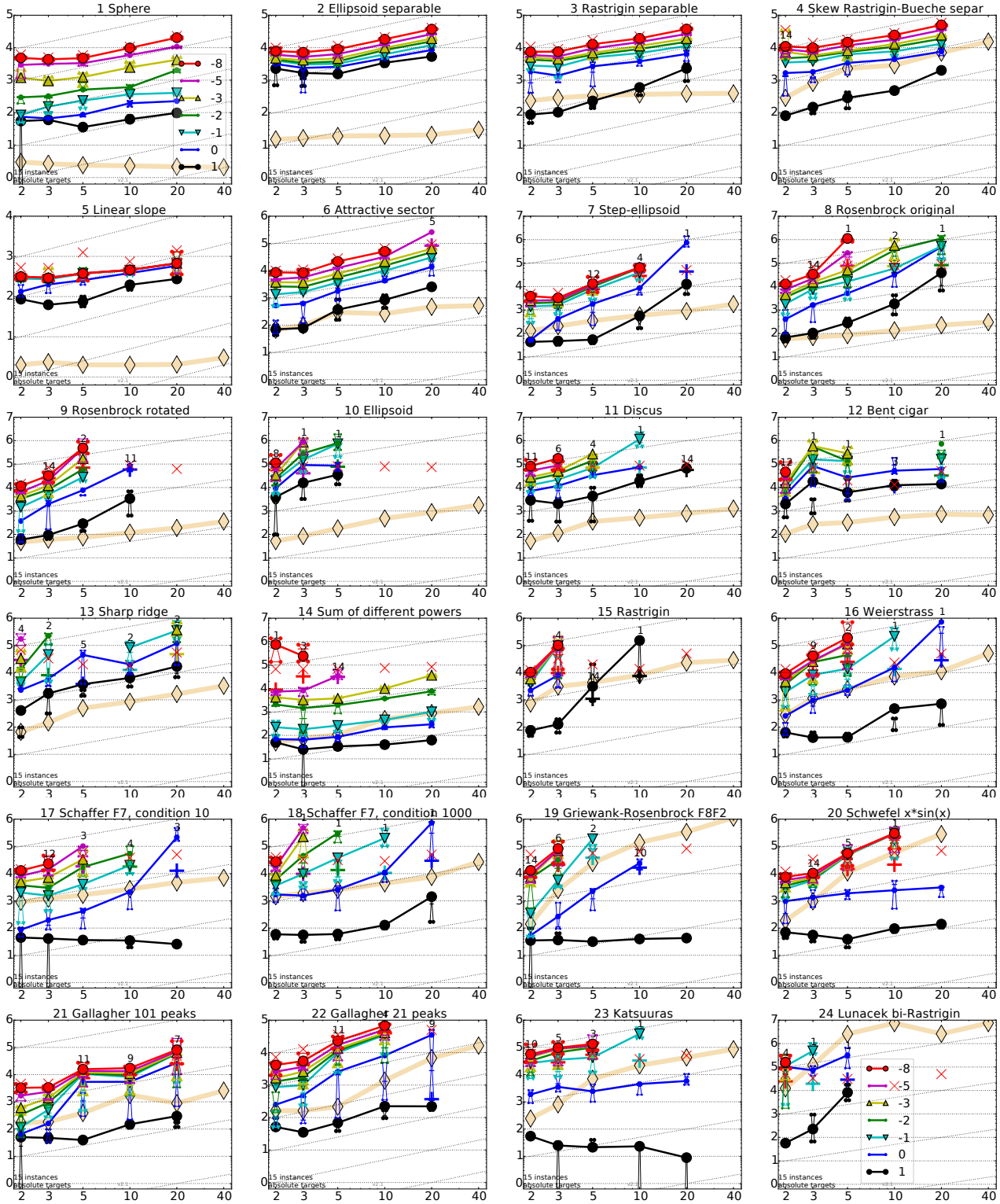


Figure 1: Scaling of runtime with dimension to reach certain target values Δf. Lines: average runtime (aRT); Cross (+): median runtime of successful runs to reach the most difficult target that was reached at least once (but not always); Cross (×): maximum number of f-evaluations in any trial. Notched boxes: interquartile range with median of simulated runs; All values are divided by dimension and plotted as $\log_{10}$ values versus dimension. Shown is the aRT for fixed values of $\Delta f = 10^k$ with k given in the legend. Numbers above aRT-symbols (if appearing) indicate the number of trials reaching the respective target. The light thick line with diamonds indicates the best algorithm from BBOB 2009 for the most difficult target. Horizontal lines mean linear scaling, slanted grid lines depict quadratic scaling.

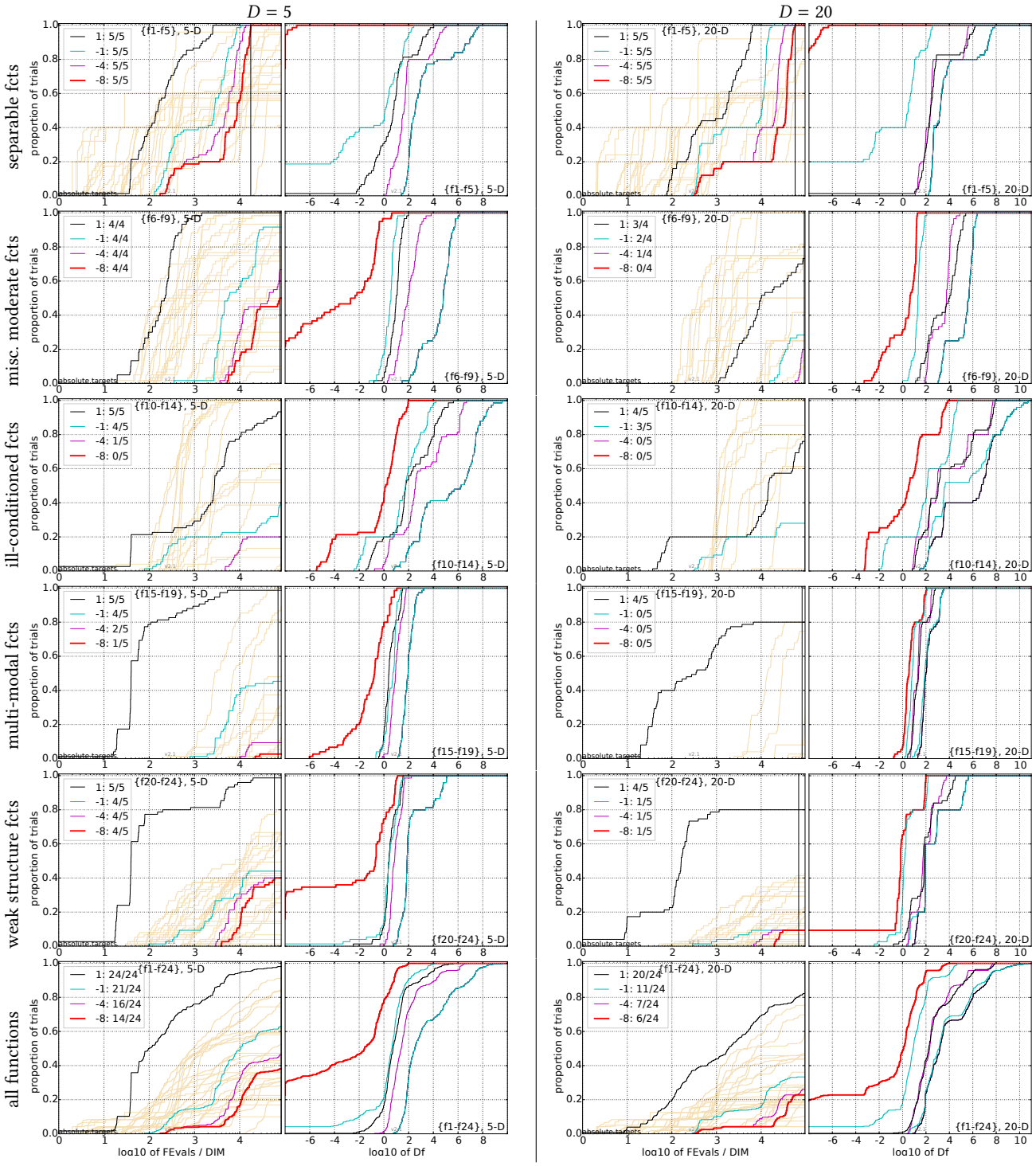
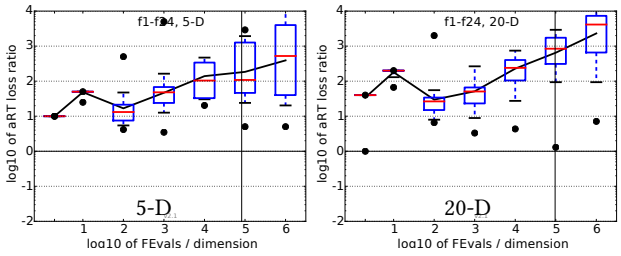


Figure 2: Empirical cumulative distribution functions (ECDF), plotting the fraction of trials with an outcome not larger than the respective value on the x-axis. Left subplots: ECDF of the number of function evaluations (FEvals) divided by search space dimension D, to fall below $f_{\text{opt}} + \Delta f$ with $\Delta f = 10^k$, where k is the first value in the legend. The thick red line represents the most difficult target value $f_{\text{opt}} + 10^{-8}$. Legends indicate for each target the number of functions that were solved in at least one trial within the displayed budget. Right subplots: ECDF of the best achieved Δf for running times of $0.5D, 1.2D, 3D, 10D, 100D, 1000D, \ldots$ function evaluations (from right to left cycling cyan-magenta-black…) and final Δf-value (red), where Δf and Df denote the difference to the optimal function value. Light brown lines in the background show ECDFs for the most difficult target of all algorithms benchmarked during BBOB-2009.



f_1–f_{24} in 5-D, maxFE/D=82734						
#FEs/D	best	10%	25%	**med**	75%	90%
2	10	10	10	10	10	10
10	25	50	50	50	50	50
100	4.2	4.8	7.4	13	22	95
1e3	3.5	11	23	48	69	2.1e2
1e4	20	30	32	1.0e2	3.4e2	6.3e2
1e5	5.0	17	44	1.1e2	1.3e3	2.0e3
1e6	5.0	18	40	5.2e2	4.1e3	1.2e4
RL_{US}/D	2e4	2e4	2e4	2e4	6e4	7e4

f_1–f_{24} in 20-D, maxFE/D=96310						
#FEs/D	best	10%	25%	**med**	75%	90%
2	1.0	40	40	40	40	40
10	67	1.0e2	2.0e2	2.0e2	2.0e2	2.0e2
100	6.6	7.8	14	26	34	2.5e2
1e3	3.3	6.3	23	51	67	3.4e2
1e4	4.3	26	1.0e2	2.4e2	4.0e2	1.2e3
1e5	1.3	83	3.1e2	8.4e2	1.8e3	3.9e3
1e6	7.1	84	5.8e2	4.1e3	7.8e3	1.7e4
RL_{US}/D	5e4	5e4	5e4	5e4	6e4	8e4

Figure 3: aRT loss ratio versus the budget in number of f-evaluations divided by dimension. For each given budget FEvals, the target value f_t is computed as the best target f-value reached within the budget by the given algorithm. Shown is then the aRT to reach f_t for the given algorithm or the budget, if the best algorithm from BBOB 2009 reached a better target within the budget, divided by the aRT of the best algorithm from BBOB 2009 to reach f_t. Line: geometric mean. Box-Whisker error bar: 25-75%-ile with median (box), 10-90%-ile (caps), and minimum and maximum aRT loss ratio (points). The vertical line gives the maximal number of function evaluations in a single trial in this function subset. See also Figure 4 for results on each function subgroup.
Data produced with COCO v2.1

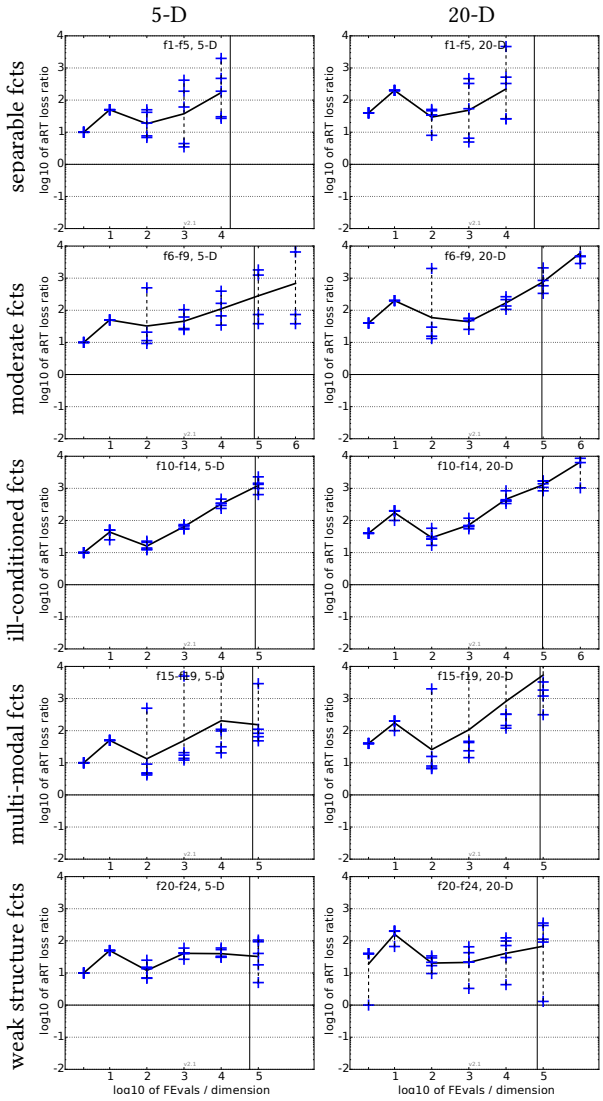


Figure 4: aRT loss ratios (see Figure 3 for details). Each cross (+) represents a single function, the line is the geometric mean.