Auto-tuning of IO accelerators using black-box optimization

Sophie Robert*†, Soraya Zertal†, Gaël Goret*
†Li-PaRAD, University of Versailles
*Atos BDS R&D Data Management
Email: {sophie.robert2, soraya.zertal}@uvsq.fr, {sophie.robert, gael.goret}@atos.net

POSTER PAPER

Abstract— **High Performance Computing (HPC) applications' performance and behavior rely on software and hardware environments which are often highly configurable. Finding their optimal parametrization is a very complex task. The size of the parametric space and the non-linear relationship between the parameters and the delivered performance make hand-tuning, theoretical modeling or exhaustive sampling unsuitable for most cases. In this paper, we propose an auto-tuning loop that uses black-box optimization to find the optimal parametrization of IO accelerators for a given HPC application in a limited number of iterations, without making any assumption on the performance function. After a literature review of the selected methods for tuning the accelerators, we describe their implementation and experimentation in our HPC context using two IO accelerators developed by Atos. We also define several metrics to evaluate the quality of our optimization, as our criteria of success go further than finding the optimal parameters. The obtained results show that this framework successfully improves the execution time of two applications used conjointly with a pure software accelerator and a mixed hardware-software one. We indeed observe possible time gains of respectively 38% and 20% for each accelerator compared to launching the same application accelerated with the default parameters.**

Keywords: Input/output; optimization; auto-tuning; performance; black-box optimization

I. INTRODUCTION AND CONTEXT

Future Exascale computers need both software and hardware intelligent components, with auto-adaptive capabilities, able to consider their environment variations to provide and maintain an optimal performance. This is especially true for HPC applications, known for their high computing consumption, their huge amount of data and their complex access profiles.
Managing such data to ensure an optimal execution is challenging and tools are needed to investigate the runtime behavior of applications, in order to understand and resolve the possible bottlenecks. IO accelerators are among these tools and can range from pure software, like the Atos Fast IO Libraries, to mixed hardware/software accelerators, such as the Atos Flash Accelerators, which can be inserted in the IO data path to accelerate IO on specific files for target applications [1]. These flash accelerators can either be used as transparent caching on a file by file basis for selected applications, a feature called Smart Burst Buffer, or as a static on demand allocation of fast persistent storage per applications' request, a feature called Smart Bunch of Flash. These tools have a significant impact on the IO execution performance depending on the interaction between their parameters and the applications' IO profiles, but their parametrization is hard to achieve manually.
A way to emancipate from the complexity of manually tuning the IO accelerators is to optimize the performance without making any assumption on the impact of the parameters. One of the promising methods to achieve this auto-tuning is called black-box optimization [2] which provides a way to find the optimal value of a function without making any hypothesis on its behavior and more particularly on the relationship between the parametric space and its value.

We have adapted these methods for our use-case to build a feedback loop (see Figure 1) that iteratively adjusts the accelerators' input parameters according to the application's execution time, so that the effect of the accelerator on the job is continuously improved.

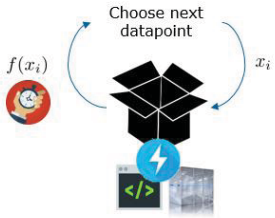

Figure 1: Schematic view of the auto-tuning loop

In addition, as we are using these methods for active tuning and not just for optimization, we have defined specific metrics to evaluate the quality of the algorithms under such constraints.

In the remainder of the paper, section II introduces the theoretical methods from the literature we used, as a background for a better understanding of our proposition. Then, we present in section III our specific environment and tools and how we intend to accelerate the applications execution time using the auto tuning framework. Section IV describes the conducted tests to validate our methodology with both the experimentation environment and process. It is followed by the

presentation of the obtained results and their discussion in section V. Finally, we conclude our paper and give some hints for future works in section VI.

II. THEORETICAL BACKGROUND

Many theoretical materials exist in the literature and we selected the heuristics the most appropriate to our context and its constraints. We give in the following sections a brief overview of these materials.

A. Black-box optimization

Black-box optimization refers to the optimization of a function of unknown properties, most of the time costly to evaluate, which entails a limited number of possible evaluations [2]. The goal of the procedure is to find the optimum of a function *f* in a minimum of evaluations without making any hypothesis on the function.

More formally, given a budget of a maximum of *n* iterations, we want to get as close to the true optimum of the function as possible:

$$\text{Find } \{x_i\}_{1\leq i\leq n} \in \Theta \text{ s.t. } |min(\{f(x_i)\}_{1\leq i\leq n}) - min(f) \leq \epsilon|$$

with:

- *f* the function to optimize (black-box)
- Θ the parameter space
- ε a convergence criterion between the found and the estimated minimum

The first step of any black-box optimization algorithm is the selection of the initial parameters to start the optimization process. An acceptable initialization starting plan must respect two properties [3] [4]: it should respect the space's constraint and the non-collapsible property.
The space constraints are shaped by the possible values that can be taken by the parameters. The non-collapse property specifies that no parametrization can have the same value on any dimension. It ensures that if an axis of the parametric space is removed, then no two points would have the same coordinates. This is especially important as we have no insight on the individual effect of each parameter.

B. Selected heuristics

Many initialization strategies and optimization heuristics are available in the literature to perform black-box optimization. We focus on only a subset of possible algorithm, selected by both their efficiency proven in previous papers, the simplicity of their implementation and their execution time.

1) Surrogate models

The main idea behind surrogate modeling is to use a regression or interpolation technique over the currently known surface to build the surrogate and to then select the most promising data point in terms of performance gain using this surrogate function [4] [5].
Any regression or interpolation function can be used to build the surrogate, if it provides a computationally cheap approximation of the black-box function.
The simplest acquisition method is to use the surrogate as the cost function and find its optimum using an optimization algorithm. This method has the advantage of not making any statistical assumption on the black box function.
Other acquisition functions require that some statistical assumptions are made on the black-box function: it is the case when using the *Maximum Probability of Improvement* (MPI) function and the *Expected Improvement* (EI) function.

The MPI acquisition function is the probability that the execution time for a parametrization will be lower than the current best execution time. Under the assumption that the black-box function can be accurately described by a Gaussian process [6], the estimated mean and standard error for each data point is available and we can derive a closed form for the MPI and select its minimum. While this method encodes the probability of getting a better parametrization, it does not encode the possible time gain from this parametrization: the function encoding this gain is the *Expected Improvement* function. It computes the expectation of the improvement function, that can be defined by the difference between the predicted execution time and the current optimum. As is the case for the MPI function, a closed form function can be derived and minimized.

2) Simulated annealing

Introduced in [7], the simulated annealing heuristic is a hill-climbing algorithm which can probabilistically accept a worse solution than the current one. The probability of accepting a value worse than the current one decreases with the number of iterations. The algorithm is started using an initial "temperature" value, which will decrease over time (hence the analogy with metal annealing). Then, at each iteration step, the algorithm randomly selects a parametrization neighboring the current one, obtains the execution time, and then makes a choice according to the new value. If the new parametrization is better than the current parametrization, it is automatically accepted. If not, a probability of acceptance is computed. If this probability is lower than a number randomly drawn from the uniform distribution, the new value is accepted even though it is worse than the current state. This makes sure that the algorithm can move out of a local minimum if it gets stuck in one. The temperature is then reduced one step, according to a given function, called a cooling schedule. This ensures that as the temperature cools down, the probability of accepting a worse solution than the current one gets lower. At the end of the algorithm, as the temperature looms close to zero, the probability of acceptance of a worse solution draws to zero.
The cooling schedules are functions of the iteration step and the initial temperature, which decrease to zero as the iteration step increases.

3) Genetic algorithms

Genetic algorithms consist in selecting two combinations of parameters among the set of already tested parametrization, according to a selection process that considers the fitness of each parent [8]. These two parameters are then combined to

create a new one. This newly created combination can undergo a mutation, which subtly alters it to provide a new one.
Many methods exist to select the two fittest combinations that should be mixed into a new one [9]. Crossover is the method by which the two combinations will merge to create a new one. The most common method, as inspired by biology, is to use single-point crossover, which consists into randomly splitting each parametrization in two and concatenating the two parents. Variants of this technique are called n-points crossover and consist in cutting the parents into *n* parts and alternatively concatenating them. We experimented with single and two points crossovers. Mutation is a random event that can happen to the offspring and modify some of its values. This ensures some randomness in the optimization process. The probability of the event happening is called the mutation rate.

III. AUTO-TUNING FRAMEWORK

We describe in this section how we adapted the previously described methodologies and methods to our specific context of IO for HPC applications. We will give details on the parametrization of such methods to build the tuning framework immediately included into the Atos IO execution chain. We also introduce the metrics designed to compare the heuristics and rank them according to their usefulness to achieve the auto-tuning.

A. Defining the black-box

In our case, we define the black-box as a combination of the accelerator, the application and the topological hardware and software context. The input parameters are the IO accelerator's parameters which are the only ones we have control over. Setting these parameters is essential as it gives us the very first information about the application and the accelerator. In our case, all the tunable parameters are bounded integers and respect the non-collapse property. We adopted Latin Hypercube Designs (LHD) [3] to use as starting designs for our methodology. When given these parameters, the black-box outputs the corresponding application execution time, which is the value to be optimized. The black-box optimization algorithm then uses a heuristic to select the next data points depending on the previous parametrization and their corresponding output.

This creates a feedback loop which continuously improves the application's execution time as shown in Figure 1.

B. Implementation of the heuristics

We experimented with the three heuristics described in section II.B, adapting them to the context of the parametrization.

1) Surrogate models

We used gaussian process regression process to perform the regression of the function and compared the results yielded by the three acquisition methods described in subsection II.B.
When using either MPI or EI, we simply evaluated the function over the whole grid and selected the maximum value (brute force maximizer).

2) Simulated annealing

The simulated annealing algorithm requires a neighboring function. We settled for a random walk in every direction of the parametric space: at each iteration, the algorithm can increase or decrease by one unit each possible parametrization.
Three cooling schedules were tested: an exponential schedule, as introduced in the original paper [7], and a multiplicative and logarithmic one. The initial temperature value was set to 1000 considering the iteration budget size.
We also consider the possibility of restarting the system, by randomly resetting the system's temperature back to its original value and resetting it once the system's energy has gone under a given threshold.

3) Genetic algorithms

Two methods were compared for the selection of the fittest parametrizations: the probabilistic pick, which randomly picks the fittest parametrizations according to their execution time, and the tournament pick which randomly divides the already tested parametrizations in two and select the one yielding the best execution time in each pool. Crossovers were made using double point crossovers when the size of the parametric space allowed it. In case of a smaller sized space, a single point crossover was used.
As this method also require defining neighboring parametrizations, we opted for the same method as with simulated annealing: a parametrization can mutate into another one with a random walk.
We set the random Bernoulli law parameter which randomly triggers the mutation to a probability of 0.1.

C. Defined metrics

Comparing the different methods requires the definition of metrics that can be used to rank the different heuristics according to their relevance and usefulness to reach our goal of performing auto-tuning of IO accelerators.
The most obvious metric is the distance between the optimum value found by the algorithm and the true optimum. While this metric is essential when evaluating the performance of an optimization algorithm, and almost sufficient if the only constraints on the algorithm is the quality of the results it yields, it is not enough in our case because we want consistency and stability in the results. Our software shipped with the IO accelerator cannot provide too many regressions as this would make it unusable for the end-user. We are thus looking for a trade-off between the quality of the returned optimum and the stability of the results yielded at each iteration.
To account for all our constraints, six metrics were defined to rank the algorithms. The three first ones (S, IS, Dist Opt) evaluate their capability at finding the true optimal value within a finite budget, by measuring their success, the speed at which they can find it and the distance between the returned optimum and the true one. Two other metrics (Avg Dist, EC) are designed to evaluate the consistency of the function value at each step, by respectively averaging the distance between the current fitness value and the true optimum value and summing up the losses due to regressions (i.e. when the application performs

worse than it previously did). The last metric (ET) consists in the time spent by the heuristic to terminate its iterations budget. These metrics were measured on several instances of each heuristic, to average any effect that could be linked to the random components of the algorithm (the initialization strategy, the selection of nearest neighbors, …).

Table 1: Description of the metrics

Metric name	***Abbr.***	***Description***	***Field specific interpretation***
Success	S	Whether or not the correct optimum was selected	Whether or not the algorithm optimally parametrized the accelerator.
Number of iterations to reach success	IS	When the correct value was reached, the convergence speed to this value	The number of iterations the user has to wait to launch the application with an optimal execution time.
Distance between optimum and correct value	DistOpt	The distance between the guessed optimum and the true optimum	The difference between the optimal execution time the user could have had and the actual returned optimal parametrization.
Distance between current selected value and correct value averaged over all iterations	AvgDist	The mean distance between the correct value and the value returned at each step by the algorithm	The performance loss experienced by the user because of being in a suboptimal state.
Exploration cost	EC	The number of regressions and their summed difference	The number of times the client sees its application performing worse than he has already seen it.
Elapsed time	ET	The total computation time	Time spent by the loop for computing the next solution.

IV. TESTS AND VALIDATION

To test and validate the methodology described above, we performed two experiments on two different accelerators, a software accelerator called Small Read Optimizer included in the Fast IO Libraries and the Smart Burst Buffer. In each experiment, we repeatedly launched an IO benchmark application with different combination of parameters. The execution times and the corresponding parametrizations are stored in a dataset. As the use case for these accelerators widely differ, we picked the test applications accordingly.

This previous collection phase enables the simulation of the auto-tuning loop by mimicking the real-life computer system (i.e. the black-box in Figure 1). At each iteration, we query for the parametrization selected by the heuristic the corresponding execution time that was collected during the experiment. This enables us to evaluate and characterize the ability of the heuristics to tune the accelerators. On each of these datasets we ran several variants of each heuristics, collecting the values for the metrics described in subsection III.C.

A. Experimentation environment

All the code used to perform the black-box optimization and collect the results was written in Python and packaged in a general-purpose library. It has been designed to be agnostic of its usage. Note that all algorithms were built from scratch with no dependence on external libraries other than the *numpy* [10] and the *sklearn* [11] library.

The validation datasets were built using an in-house automatic application and HPC context manager, also written in Python. This software enables the launch of many repetitions of the same application with different parametrizations of the accelerator and monitors the execution time and the IO behavior of the scheduled jobs.

All the experiments were run on isolated nodes and storage systems to minimize potential interferences.

B. Conducted tests

1) Smart Burst Buffer (SBB)

To conduct the tests for the tuning of the smart burst buffer, we used an IO benchmark application performing two write checkpoints consisting in three parallel write processes with 5000 random writes. Each write is achieved in a different file and covers 250KB. These two checkpoints are separated by a 10 seconds sleep. We tested 681 different parametrizations, in a parametric space with four dimensions.

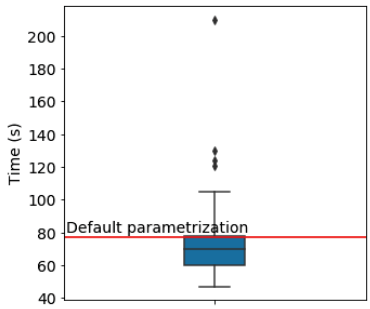

Figure 2: Statistical repartition of execution time using several parametrizations of the SBB accelerator

2) Small Read Optimizer (SRO)

We tested 264 parametrizations of the Small Read Optimizer accelerator on a IO benchmark application performing 200000 reads of 4KB each using a stride pattern. For this accelerator, the parametric space has three dimensions.

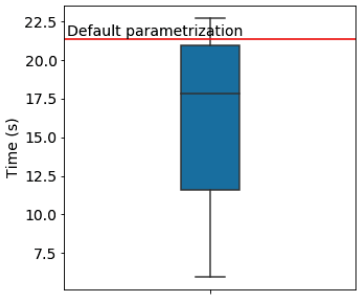

Figure 3: Statistical repartition of execution time using several parametrizations of the SRO accelerator

We can see that for both datasets the default parametrization shipped with the accelerator is not optimal and preliminary results have shown that the optimal parametrization varies depending on the application.

V. Results and Discussion

For each variant of the heuristic we used for our auto tuning methodology, and for each dataset, we computed the defined metrics introduced previously. Each heuristic is run 20 times to average random behaviors and each algorithm has a total budget of 100 iterations. We summarize the obtained results in Table 2 below.

Table 2: Evaluation metrics for each heuristic on the test datasets

Accelerator	Heuristics	Metrics					
		S (%)	*IS*	*Dist Opt (s)*	*Avg Dist (s)*	*EC (s)*	*ET (s)*
SBB	SM	100	82	0	25.91	2398	48.2
	GA	40	43	4.4	12.85	984	15.8
	SA	20	47	7.1	19.19	1220	12.1
SRO	SM	0	0	0.1	9.14	938	31.5
	GA	10	20	5.4	8.35	279	12.6
	SA	0	0	8.9	9.32	305	10.5

The first result is that in both cases (SBB and SRO), surrogate models perform best in terms of proximity to the true optimal parametrization. In the case of the SBB, the minimum has been found in all the 20 optimization paths whilst in case of the SRO, even though no trajectory has succeeded at finding the optimum, the returned optimal parametrization yields an execution time 1ms lower than the actual optimum.

However, as was described in section III.C, our criteria of success go further than finding the optimal parameters: the stability of the execution time observed by the user is an essential feature of the loop. We find that genetic algorithms offer the best trade-off between accuracy and stability, with an average difference of 12.85 seconds (SBB dataset) and 8.35 seconds (SRO dataset) compared to the optimum and an exploration cost of 984 seconds (SBB dataset) and 279 seconds (SRO dataset). Their returned optimum is close to the optimal

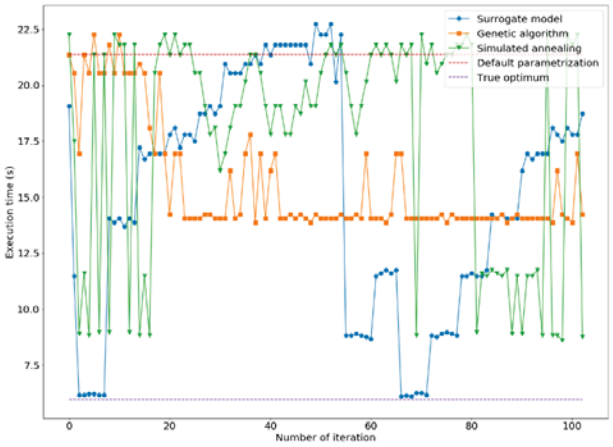

Figure 4: Execution time for each heuristic tuning the Small Read Optimizer

value in both cases (4.4 seconds for the SBB and 5.4 seconds with the SRO).

Even though the surrogate models perform better at returning the optimum fitness, their switching between values is too erratic. The same can be said about simulated annealing, which has a high exploration cost and a low accuracy. These results are a good demonstration of the trade-off between exploration and exploitation and can be explained by the nature of the algorithms: once the surrogate models have properly explored a subset of the parametric space (i.e. the variance

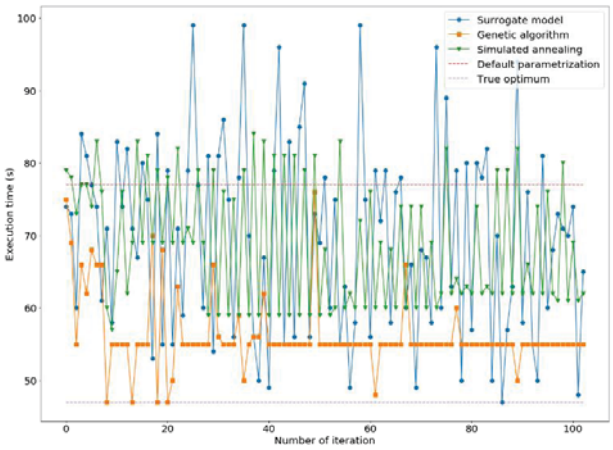

Figure 5: Execution time for each heuristic tuning the Smart Burst Buffer

estimated by the gaussian process decreases) it moves on to zones with higher variances. The genetic algorithm does not display a similar behavior as its stochastic behavior is limited to mutation and can only switch to a neighboring parametrization.

Figure 4 and Figure 5 show two examples of optimization trajectories which illustrate our interpretations of the results. They represent the execution time observed by the user at each iteration of the tuning loop. In Figure 4, we can see that the surrogate models loom very close to the optimum value but moves on to unexplored space with longer execution times. The same can be said in Figure 5 in which surrogate models oscillate between close to optimum and slower execution time. Simulated annealing exhibits a similar behavior. We insist that such a behavior is not acceptable in an HPC production setting. In the two figures, genetic algorithms exhibit a more stable behavior. Even though it remains in a suboptimal state, it performs better than the default parametrization and does not cause any major performance regression.

Overall, the tuning framework improves the execution time compared to the default parametrization. Indeed, averaged over all heuristics and all trajectories, 77.57 percent (SBB dataset) and 89.47 percent (SRO dataset) of the execution time returned by the loop are below the execution time yielded by the default parametrization. 51.24 percent (SBB dataset) and 59.25 percent (SRO dataset) of trajectories are below the mean execution time at all iteration. In both cases, genetic algorithms provide the best results when compared with respectively the default and the

mean execution time. The mean time gains over the whole loop are compared to the default parametrization are summarized for each heuristic in Table 3.

Table 3: Averaged percentage of time gained compared to default parametrization over the 100 iterations for each heuristic

Time gain compared to default parametrization (%)	Heuristics		
	SM	*GA*	*SA*
SBB accelerator	9.8	20.5	13.1
SRO accelerator	30.1	38.4	21.3

These results validate the relevance of black box optimization in our HPC specific context. Efficiency was observed for both datasets which describe the behavior of two very different IO-oriented applications with two very different accelerators, the SRO being a software accelerator and the SBB a mix between hardware and software.

VI. Conclusion and Future Works

In conclusion, we have successfully demonstrated the usefulness of using black-box algorithms for auto-tuning a pure software and a mixed soft-hardware IO accelerator in the HPC context.

To improve upon this work, we are planning on considering a wide variety of applications with different IO profiles and larger ranges of parametrizations.
One of the major next steps is investigating the robustness of black-box optimization algorithms in a stochastic context. For now, we have operated in a deterministic context, while our cluster have a stochastic behavior. In fact, even in a controlled environment, launching the same job with the same parametrization does not return the same value. Asserting the impact of noise and adapting the algorithms accordingly is unavoidable for their use in a production setting.
Finally, as a long-term goal, we will consider the optimization process across many applications by considering their IO behavior.

Acknowledgement:

This work has been partially funded by the ASPIDE Project funded by the European Union's Horizon 2020 research and innovation program under grant agreement No 801091.

Références

[1] [Online]. Available: https://atos.net/fr/produits/calcul-haute-performance-hpc/extreme-data.

[2] L. M. Rios and N. V. Sahinidis, "Derivative-free optimization: a review of algorithms and comparison of software implementations," *Journal of Global Optimization,* vol. 56, no. 3, p. 1247–1293, 2013.

[3] K.-T. Fang, R. Li and A. Sudjianto, Design and Modeling for Computer Experiments, Chapman and Hall/CRC, 2005.

[4] V. K. Ky, C. D'Ambrosio, Y. Hamadi and L. Liberti, "Surrogate-based methods for black-box optimization," *International Transactions in Operational Research.,* no. 24, 2016.

[5] L. Vincent, M. Nabe and G. Goret, "Self-optimization Strategy for IO Accelerator Parameterization," *International Conference on High Performance Computing.,* pp. 157-170, 2018.

[6] C. E. Rasmussen and C. K. I. Williams, Gaussian Processes for Machine Learning, The MIT Press, 2006.

[7] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, no. 4598, pp. 671-680, May 13, 1983.

[8] L. Davis, Handbook Of Genetic Algorithms, Van Nostrand Reinhold, 1991.

[9] N. Saini, "Review of Selection Methods in Genetic Algorithms," *International Journal Of Engineering And Computer Science,* vol. 6, no. 12, pp. 22261-22263 , 2017.

[10] S. v. d. Walt, S. C. Colbert and G. Varoquaux, " The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering,* no. 13, pp. 22-30, 2011.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher and M. Perrot, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* no. 12, pp. 2825-2830 , 2011.

[12] J.-Y. Potvin and M. Gendreau, "Handbook of Metaheuristics," *International Series in Operations Research & Management Science,* 2010.