Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

COMPUTER SCIENCE

A comparative study of black-box optimization heuristics for online tuning of High Performance Computing I/O accelerators

Sophie Robert[1,2] | Soraya Zertal[2] | Grégory Vaumourin[1] | Philippe Couvée[1]

[1]Data Management Division, Atos, Echirolles, France
[2]Li-PaRAD, University of Versailles, Versailles, France

Correspondence
*Soraya Zertal, Li-PaRAD (UVSQ).
Email: soraya.zertal@uvsq.fr

Present Address
Bat. Rabelais, 9 bd d'Alembert, 78280 Guyancourt, FR

Abstract

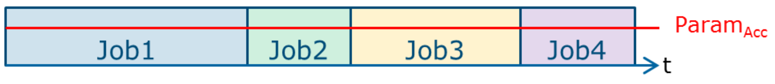
High Performance Computing (HPC) applications' behaviors rely on highly configurable software environments and hardware devices. Finding their optimal parametrization is a complex task, as the size of their parametric space and the non-linear behavior of HPC systems make hand-tuning, theoretical modeling or exhaustive sampling unsuitable in most cases. In this paper, we propose an online auto-tuner that relies on black-box optimization to find the optimal parametrization of Input/Output (I/O) accelerators for a given HPC application in a limited number of iterations, without making any assumption on the behavior of the tuned system. As many heuristics are available in the literature, we need to guarantee the quality of the tuning by selecting the most appropriate one. To do so, we provide a comparative study of the efficiency of three heuristics applied to tuning two I/O accelerators developed by the Atos company: a pure software accelerator (Small Read Optimizer) and a mixed hardware-software one (Smart Burst Buffer). To select the most efficient heuristic for our use case, we define several new metrics to evaluate the quality of an auto-tuner, in an online and offline settings. We find that genetic algorithms provide a faster convergence rate and a faster computation time but Surrogate Models provide a better score in terms of both distance to the optimum and trajectory stability. Overall, the obtained results show that auto-tuning heuristics improve the execution time of applications used conjointly with both SRO and SBB accelerators.

KEYWORDS:
Online tuning, Auto-tuning, Adaptive computer system, Black-box optimization, Surrogate models, Genetic algorithms, Simulated annealing, I/O accelerators, Performance

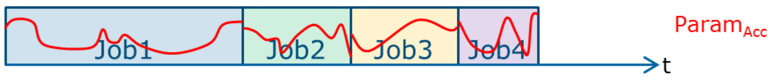
1 | INTRODUCTION

Modern computer systems expose many configuration parameters which users must tune to maximize their performance. In order to make the most of their resources and improve their performance, intelligent software and hardware components should exhibit some auto-adaptive qualities and be automatically able to select the most appropriate parameters for their current workload. This problematic is particularly important in the field of High Performance Computing (HPC) which handles heavy computation and Input/Output (I/O) workloads. As supercomputers perform a wide range of computationally intensive tasks, they require that their different components are tuned with the most adapted parametrization for the currently running application.

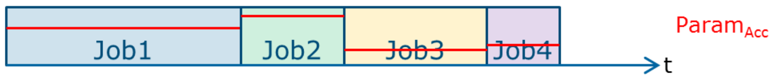
Several approaches can be considered for finding the optimal parametrization of a computer system. The first one consists in giving a static (default) parametrization for every application run by the system, as described in figure 1a. While easy to implement, it can often cause suboptimal performance because no single parametrization can match the requirements of a great diversity of workloads. Another solution, described by figure 1b, consists in giving at run-time different parametrization for each timeframe of the workload execution, but the optimization mechanism may compete with the application's resources and the development costs associated to the complexity of making softwares or hardwares so flexible make this solution hard to implement. The last approach consists in selecting a parametrization adapted to each running application and keep it constant throughout the run, as described in figure 1c. This type of parametrization, called adaptive, is a trade-off between dynamic and static parametrization [1].



(a) Static parametrization



(b) Dynamic parametrization



(c) Adaptive parametrization

FIGURE 1 Different possible parametrization approaches

Several strategies can be adopted to find this optimal adaptive parametrization for a given application. The first solution could be to perform an exhaustive sampling of the parametric space by testing for each parametrization its impact on the performance function of the application. However, the large size of the parametric space and the long duration of most HPC applications makes this option too time consuming and thus impractical in most cases. An alternative could be to design a theoretical explicit model for the software or hardware to be tuned which would describe the relationship between the parameters, the application and the performance function. However, building such a model is difficult because of the complexity of the relationship between each component of a computer system and the lack of insight on the system's behavior. Also, this model would only be usable for this particular component and would not be suitable for another context. Auto-tuners relying on black-box optimization have emerged as a solution for finding the optimal parametrization without tuning the system manually, nor through theoretical or field expertise insights. It consists in a black-box optimizer iteratively evaluating configurations to find an efficient one for a given workload. By their nature, black-box optimization tuners are agnostic of the system under-going tuning and do not require any hypothesis on the parametric space.

The parametrization can take place *offline*, to find the optimal parametrization for the workload without the system running in production. This can be considered as a setup phase as described in [2], which consists in finding the optimal parametrization for a given component when installing the machine. Most tuners described in the literature are designed for offline tuning. We suggest the other possibility of performing the tuning *online*, by using the auto-tuner in production. This adds several constraints on the convergence guarantees of the tuner which we will detail in the coming sections.

There are many definitions with different shades of meaning to name the different types of tuning in the literature. We focus here on the autotuning and more precisely on the online autotuning. In [3], autotuning refers to the automatic generation of possible implementations which are evaluated from models or empirical measures which constitute the search space to identify the most desirable implementation. In our case, a possible implementation is one I/O accelerator's parametrization and the most desirable one is the one which performs better and the considered heuristics (SM, SA, GA) are the generators of the search space. When it comes to the online autotuning, also called incremental autotuning, it is the one which occurs over one or multiple runs of an application [3], which is exactly what happens on our tuned system. We can find also that the online dynamic autotuning is conducting measures during the execution, gathering these information and adapt the system during the execution. We call this just dynamic autotuning and we discarded it because of its cost in term of compute resources having a negative impact on the execution of the user application which is very penalizing in HPC context. Our definition of the online autotuning is running experiments offline and gathering knowledge from the input data points and the associated measured performance function. This means parametrizations of the IO accelerator and related applications execution time in our case. It is achieved offline and a database of this knowledge is built accordingly. Then, every user application submitted to the system, goes through a recognition engine based on meta-data which try to match this application with previously launched ones. If the matching process is positive, previous runs of this application-like are used to give its parametrization, otherwise the setup is created using initialization techniques. The term online comes mainly from this recognition of the incoming application and also from the continuously knowledge construction along with the applications executions.

This paper is based on our early paper [4] and describes an online adaptive auto-tuner which finds for a given HPC application the optimal parametrization of an I/O accelerator designed for the HPC context. I/O accelerators, which can either come as software or hardware components, intend to speed-up the I/O performed by HPC systems and fill the ever-increasing performance gap between the compute nodes and the storage bays. They are a critical tool for the next generation of HPC systems aiming to solve the I/O bottlenecks problem. Because black-box optimization is oblivious to the optimized system and because it performs well on sparse data, it is a good candidate for auto-tuning of computer systems, including I/O accelerators.

We suggest to include a feedback loop as shown in figure 2 in the workflow to iteratively adjust the accelerators' input parameters according to the application's execution time, so that the effect of the accelerator on the job is continuously improved. The parameters of the I/O accelerator are autotuned while the application is executed in a production environment by a user, instead of performing blocking offline autotuning and then the production execution of the application. Therefore, the autotuning framework requires the user to execute the application multiple times with the input which does not change the application runtime itself (e.g., the input of the same volume if the input volume is sufficient to define application runtime).

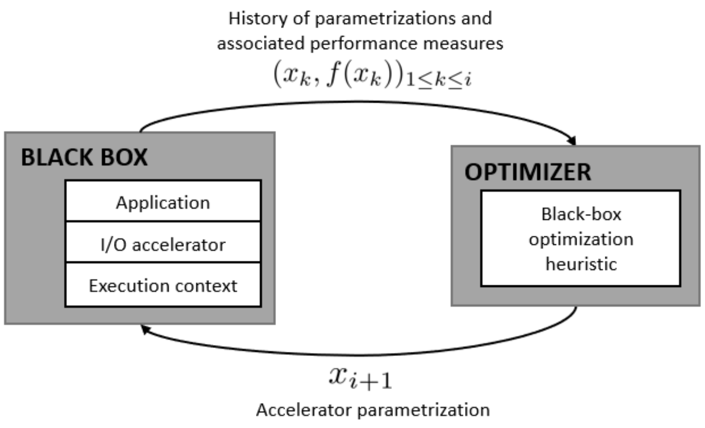

History of parametrizations and associated performance measures
$(x_k, f(x_k))_{1\leq k\leq i}$
BLACK BOX
Application
I/O accelerator
Execution context
OPTIMIZER
Black-box optimization heuristic
x_{i+1}
Accelerator parametrization

FIGURE 2 Schematic view of the auto-tuning loop

In our suggested workflow, an application recognition engine, relying on meta-data matching, tries to match an incoming application with previously submitted applications. If this application matches any already known submissions, the optimization process uses previous experiments of this application to automatically setup the experiment. If not, a new entry for the application is created and the parametrization is setup using initialization techniques. Each application is launched with one and unchanged parametrization during all its execution. Once the application is running, the system accumulates knowledge about the impact of the accelerator on the application given its I/O behavior.

This builds the feedback loop that iteratively adjusts the accelerators' input parameters according to the application's performance (which in our case is the execution time), so that the effect of the accelerator on the job is continuously improved. This workflow is schematically described in section 5.1.

The black-box optimizer relies on optimization heuristics, and the literature provides a variety of them [5]. The first challenging point in this work is the choice of the most suitable one(s) in terms of convergence speed and the stability of the optimization process. This choice is based on a comparative study conducted on a HPC cluster using two targets I/O accelerators developed by the Atos company.

We propose new metrics to evaluate the relevance of three state-of-the-art black-box optimization heuristics in an offline and online production setting.

The major contributions of this paper are:

- The original use case of using black-box optimization algorithms to find the optimal parametrization of I/O accelerators, which is usually achieved by analytical performance models, using closed-form expressions for predicting performance metrics;
- The comparative study of black-box heuristics as a guideline for an appropriate selection and implementation;
- The definition of new metrics that describe the behavior of auto-tuning algorithms when used in an online setting;
- The application of black-box optimization on two very different I/O accelerators proving the universality of the method.

In the remainder of the paper, section 2 presents different works related to the optimization of I/O accelerators and more generally to the auto-tuning of computer systems. I/O accelerators and the particular constraints of our tuner are described more in depth in section 3. Section 4 introduces the theoretical grounds of black-box optimization and describes the three state-of-the-art heuristics selected for our comparative study. In section 5, we present the auto-tuning framework and its general workflow. Section 6 describes the conducted tests to validate our methodology with both the experimentation environment and process. We also detail in this section a set of new metrics designed to characterize the behavior and evaluate the quality of the auto-tuner in an online and offline settings. It is followed by the presentation of the obtained results and their discussion in section 7. Finally, we conclude our paper and give some hints for future works in section 8.

2 | RELATED WORKS

Auto-tuning using black-box optimization has been used in several domains in the last years. It has yielded good results in very diverse situations and has been particularly helpful in computer science for finding optimal configurations of various software and hardware systems. In [6], the authors compare two derivative-free methods (Bounded Optimization BY Quadratic Approximation method and Constrained Optimization BY Linear Approximation method) to find the optimal configuration of the Hadoop framework. Bayesian optimization has also been successfully used for finding the optimal configuration of Apache Storm computation system in [7]. A more general tuning framework, called BOAT, relying on structured Bayesian optimization is described in [8].

Within the HPC community, auto-tuning has gained a lot of attention for tuning particular HPC application and improve their portability across architectures [9]. In [10] and [11], a comparison of several random-based heuristic searches (Simulated annealing, genetic algorithms …) are provided when used for code auto-tuning while [12] has had good results with surrogate modeling using boosted regression trees. The HPC energy system has also benefited from Bayesian Optimization, as [13] made the Green500 list after using an auto-tuner based on a combination of Gaussian Process regression and the Expected Improvement acquisition function.

More generally, the I/O community has successfully tuned storage systems for particular workloads by using black-box optimization tuners, as storage systems are also highly configurable with a very large parametric space [14] [15]. Black-box optimization has had good success for tuning storage systems when faced with different workloads and [16] proposes a thorough analysis of the behavior of each heuristic. Reinforcement learning has also been successfully used as an auto-tuner to optimize the performance of the Lustre filesystem in data center storage systems [15] and an optimal parametrization for the several layers of HDF5 file system was found using genetic algorithms in [2]. An extension of this auto-tuner which selects the best parameters according to the I/O pattern is described in [17]. However, to our knowledge, no such method has been used in an online setting and particularly no one has suggested to use the metrics we introduce and describe in section 6.1 to quantify the cost of an auto-tuner when used in production.

When it comes to I/O accelerators parametrization and more generally the understanding of their behavior, [18] has suggested to use a Markov chain based model to describe the behavior of the burst buffer. [19] proposes a polynomial time algorithms for the special case of finding the optimal buffer size, both when considering static and dynamic resource allocation. While very successful for describing a particular conception of a burst buffer architecture, this model lacks flexibility and does not take into account the specificity of each commercial burst buffer's implementation. It is also strictly specific to the burst buffer and is not suitable for any other accelerator which can have very different effects on the I/O of a HPC application.

3 | CONTEXT OF THE STUDY: PARAMETRIZATION OF I/O ACCELERATORS FOR HPC

Before presenting our methodology and the upon-based theoretical materials, it is important to state the context of the study with its specific components and constraints, even if the proposed methodology is not problem specific and can have a broad area of use.
The final goal of this study is the optimization of the parameterization of I/O accelerators. I/O accelerators are software or hardware components whose aim is to reduce the increasing performance gap between compute nodes and storage nodes, which can slow down I/O intensive applications. On large supercomputers, the many nodes performing reads or writes stress the storage bay and can make the application wait while it performs its I/O. This phenomenon is called *I/O bottlenecks*. It is especially true for HPC applications that periodically save their current state (by performing *checkpoints*) which causes many writes during a short timeframe. The link between the compute and the storage node can become saturated, which slows down the application. To mitigate these problems, several I/O accelerators have been developed over the years. In this study, we focus on two I/O accelerators developed by the Atos company: the *Small Read Optimizer* [20] and the *Smart Burst Buffer* [21].

3.1 | The Small Read Optimizer

Accessing data *via* a cache reduces considerably the latency gap between the computing on the microprocessors and data access on the main storage system. As the data can not be entirely loaded in cache memory because of its small size, efficient cache management and an intelligent scheduling of data movements is necessary.

The Small Read Optimizer (abbreviated as `SRO`) consists of a dynamic data preload strategy to prefetch on the compute node memory, chunks of files that are regularly accessed. The main principle consists in automatically detecting zones in the file that are accessed several times in order to load into memory the whole file zone. The file is divided in several zones (`binsize`), and for each of these zones, the number of accesses is recorded. Once a certain number of access (`cluster_threshold`) has been recorded for a given bin, a zone `prefetch_size` is loaded in the cache. As this method relies on the access density per file zone, it considers both temporal and spatial aspects and is more interesting than the *read-ahead* strategy used by Linux which does not load multiple zones of a file in the same time and whose effect is restricted to sequential reads. The parametric space associated with this accelerator has four dimensions, each being positive and discrete, and is summarized in table 1.

TABLE 1 Small Read Optimizer's (SRO) tunable parameters

Name of parameter	**Description**
Binsize	The granularity of the file zones considered by SRO
Cluster threshold	The number of operations required in a given zone to trigger the prefetching mechanism
Prefetch size	Size of the zone that will be prefetched
Sequence length	The number of the previous I/O records kept by SRO

3.2 | The Smart Burst Buffer

The Smart Burst Buffer (abbreviated as SBB) is Atos' implementation of a burst buffer, an instance of a hardware I/O accelerator. It allocates on a job-by-job basis a temporary cache positioned between the computing processes and the permanent storage system. Burst of I/Os or sensitive I/Os can be redirected to this high bandwidth cache to avoid I/O bottlenecks during intensive I/O phases, such as checkpointing phases. The SBB absorbs burst of I/O and asynchronously performs the I/O on the permanent storage bay so that the job can complete sooner. On the implementation side, the SBB allocates 2 levels of caches on a specific node, called *datanode*, connected to the compute nodes through high bandwidth RDMA connections. The first cache level is a RAM cache and the second one a NVME cache. Figure 3 details a simplified view of the lifetime of an I/O going through the SBB:

1. The I/O is sent to the datanode through RDMA connection with the help of RDMA polling threads. They insert incoming I/Os into an I/O queue waiting to be treated

2. Each I/O is then inserted into the RAM cache with the help of worker threads. Reaching this point, the SBB replies to the application and the I/O is considered completed by the application.

3. The I/O is forwarded into the second level of cache built with a NVME disk with the help of RAM destagers threads (the data file is kept in the RAM cache, in case of data file reuse)

4. The I/O is finally sent to the underneath parallel file system with the help of flash destagers. The data file is kept in the NVME cache is case of data reuse.

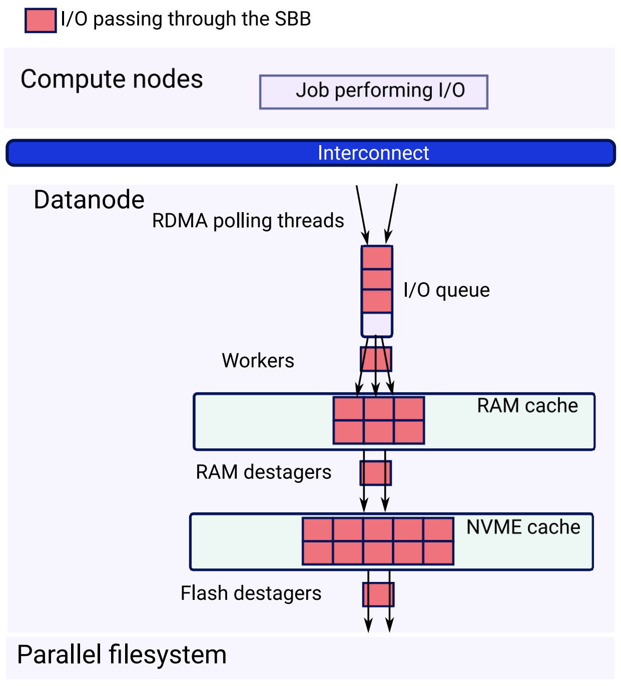


FIGURE 3 Workflow of I/Os passing through SBB

When passing through the SBB, the I/O goes through several stages, and each stage can be parameterized to allocate an appropriate amount of ressources to avoid any bottleneck in the burst buffer's workflow. The SBB parametrization is a complex problem to solve on top of being specific to the application's I/O profile. For example, an application with bursty writing behavior will require more RAM and flash destagers threads, so caches are emptied quickly enough to make sure there is always enough room in the caches to receive the next I/O and do not block the application. The considered parameter space for this accelerator has six dimensions, each positive and discrete, and are described in table 2.

TABLE 2 Smart Burst Buffer's (SBB) tunable parameters

Name of parameter	Description
Number of workers	Size of the worker threads pool used to transfer the data from the compute nodes to the Data node
RDMA Polling Threads	Number of threads used to handle RDMA transactions between the compute and datanode
Number of RAM destagers	Size of the destager threads pool used to transfer the data from the RAM cache to the NVME cache
Number of flash destagers	Size of the destager threads pool used to transfer the data from the NVME cache node to the parallel filesystem
RAM and NVME Cache size	Size of the allocated RAM/NVME cache on the Data node
RAM and NVME threshold	The fraction of used RAM/NVME cache before removing clean data from the cache to free space for the incoming I/O

3.3 | Integration of the optimizer into the I/O accelerators

The final version of the optimizer is to be integrated to the accelerators by using it along with any application submitted with it, through a Slurm [22] plugin or any other job scheduler and so every application submitted by the user will be parametrized using the optimizer. This adds several constraints to the optimizer when used in an online setting:

- *Transparency*: It must not impact the usability of the cluster. The user must get the results from his applications in a timeframe better or at least close to the usual one when not using any I/O accelerator. The optimization engine must not give the user the impression of negatively impacting the cluster's performance because of a too high variation of the accelerator efficiency (which causes a high variability in execution times) due to parametrization switch. This constraint is taken into account for evaluating the optimizer's suitability in section 6;
- *Universality*: the optimizer must be able to adapt to very different types of accelerators, application and execution context. It must also be resilient to the fact that a new accelerator can be developed at any point;
- *Sequential evaluations*: The parameters must be evaluated sequentially, and only one parametrization must be selected at each time;
- *Self-adaptive capabilities*: The model must be able to integrate new knowledge and adapt itself to the system's changes;

- *Non-reusability of results*: The optimal parametrizations are specific to the application, the accelerator and the topology. The optimum parametrization cannot simply be transposed from one use case to the next;

- *Sparsity of results*: The information given as input depends only on the user and it is not possible to submit it to any rule. For example, it is not possible to fix a number of repetitions for the application. This makes the information rare and each evaluation costly. The model must thus maximize the knowledge acquired for each run of the application.

4 | PROBLEM STATEMENT AND THEORETICAL BACKGROUND

Finding the optimal parametrization of the previously presented accelerators for a given application requires to select an optimization method among these available in the literature. Many theoretical materials exist in the literature [5] which makes the selection of the most suitable optimization method a difficult task. We selected three promising heuristics to investigate and determined the one most suited to our use case after a thorough comparative analysis. We give in this section a brief overview of black-box optimization and the selected heuristics.

4.1 | Problem formalization

As the problem of system tuning is very general, we use notations as generalist as possible in order to make our methodology problem independent. The problem can be transcribed formally as follow. Let S the system to optimize and p_i its parametrization, which belongs to a discrete subset of the possible parametrizations (which might be infinite) $\mathbb{P} = \{p_i\}_{i \in \mathbb{N}}$. Let $\mathcal{A}$ the application for which we choose to optimize the system. Let F_s be the performance function associated with the application and the system.

$$F_S : (\mathcal{A},\ \mathbb{P}) \longrightarrow \mathbb{R}$$

While in reality, F_s is a random variable which realizations are the performance measures of a particular run, it will be considered as a deterministic function in this paper and its stochastic behavior will be explored in future works.

In this paper, the tuning problem consists in finding the parametrization which minimizes the performance function F_S for a set application. $\mathcal{A}$ is thus considered to be constant and F_S can be rewritten as $F_{S,\mathcal{A}}$, which will be our target to optimize. The optimization problem can be summed up as finding:

$$\text{argmin}_{p_i \in \mathbb{P}} F_{S,\mathcal{A}}(p_i)$$

In our particular case of I/O accelerator tuner, the system S to optimize is represented by the combination between the accelerator and the execution context, which corresponds for example in the HPC context to the compute nodes and the storage bay characteristics. This execution context is considered to be set throughout the experiment in order to make the system S immutable.

As specified in the introduction, several methods can be considered for solving this optimization problem, among which exhaustive sampling (*i.e.* evaluating every parametrization in $\mathbb{P}$) and theoretical modeling (*i.e.* deriving a closed form for $F_{S,\mathcal{A}}$). Because these two solutions are not adapted to the constraints described in section 3.3, we suggest to use black-box optimization techniques. For clarity reason and in order to be consistent with the black-box optimization literature, we will denote $F_{S,\mathcal{A}}$ as f when no confusion is possible.

4.2 | 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. 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. The only available information is the history of the black-box function, as the list of the inputs and the corresponding outputs.

Given a budget of n iterations, the problem can be transcribed as:

$$Find\{p_i\}_{1\leq i\leq n} \in \mathbb{P}\ s.t.\ \mid min(f(p_i)_{1\leq i\leq n}) - min(f) \mid \leq \epsilon \tag{1}$$

with:

- f the function to optimize
- $\mathbb{P} = \{p_i\}_{i\in\mathbb{N}}$ the parameter space
- ϵ a convergence criterion as the distance between the returned and the true optimum

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 [23] [24] [25]: the space's constraints 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.

The second step of black-box optimization is a feedback step. It consists in iteratively selecting a parametrization, evaluating the black-box function at this point and selecting accordingly the next data point to evaluate by using a higher procedure for searching an optimal solution in a parametric space, called an optimization *heuristic*.

Black-box optimization heuristics rely on the trade-off between exploration and exploitation, as the algorithms must keep on exploiting zones identified as being good solutions during the initialization or the optimization process and exploring the parts of the parametric space that are still unknown in order to make sure it is not stuck in a local optimum. These two criteria are complementary and must be balanced in order to achieve good optimization properties, and every black-box heuristic encodes this exploration and exploitation trade-off.

4.3 | Selected heuristics

We focus on a subset of possible black-box heuristics, selected by both their efficiency proven in previous papers and the simplicity of their implementation [26] [27] [13].

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 on this surrogate function [24] by using an acquisition function. 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 surrogate model. Other acquisition functions require that some statistical assumptions are made on the surrogate model built to represent the black-box function. It is the case when using two state of the art acquisition functions: the Maximum Probability of Improvement (MPI) function and the Expected Improvement (EI) function [24]. These two functions make the assumption that the black-box function can be accurately described by a probabilistic model, often Gaussian Process Regression or Random Forests [28], so that each data point in the parametric space can be described by a mean and a standard error.

   The MPI acquisition function computes 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 probabilistic model [29], a closed form can be derived for the MPI and selects its minimum. While this method encodes the probability of getting a better parametrization, it does not encode the possible performance gain from this parametrization. This gain is encoded by the Expected Improvement function, which computes the expectation of the difference between the predicted performance value and the current optimum. As is the case for the MPI function, a closed form function can be derived and minimized using any optimization algorithm.

2. Simulated annealing:

   Introduced in [30], 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 of the system then decreases, according to a given function, called a cooling schedule. This cooling schedule is a function of the iteration step and the initial temperature, and must decrease to zero as the iteration step increases. 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 heuristic can also include a restart mechanism, which brings the system back to its initial temperature, either randomly or when the system gets below a certain threshold temperature. This trick ensures resilience to local optimums.

3. Genetic algorithms:

   Genetic algorithms consist in selecting a subset of parameters (which is called a population) among the already tested parametrization, according to a selection process that considers the fitness of each parent [31]. These parameters are then combined to create new ones. Two combinations of parameters merge to create a new one using a crossover method. 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. These newly created combinations can randomly undergo a mutation, to subtly alter them to create a new one. This ensures some randomness and diversity in the optimization process. The probability of this event happening is called the mutation rate. We made the choice to consider only genetic algorithms with a population of size 2, which will generate a single offspring at each generation, evaluated sequentially at each iteration as described in the section 3.3.

5 | AUTO-TUNING FRAMEWORK

In this section, we present with more details the workflow of the proposed I/O accelerator auto-tuner. We also describe the implementation of the previously presented optimization process and heuristics within the auto-tuner and discuss their different hyper-parameters used for the comparative study.

5.1 | Auto-tuner general workflow

The auto-tuner follows the general workflow summarized in figure 4. The submission of an application by the user triggers the tuning mechanism. The application first goes through an application recognition engine, which tries to match the application with an already known one based on its metadata. If the application is known by the system, it is parametrized according to its previous execution history using a black-box optimization heuristic. If not, a new entry is created for the application and the optimization algorithm uses a specific initialization strategy, as described in section 4.2, to select the accelerators' parameters. Within a given application, the combination of the accelerator, the application and the topological hardware and software context are treated as a black-box. Its input parameters are the I/O accelerator's parameters which are the only ones we have control over. 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 one of the heuristic described in section 4.3 to select the next data points depending on

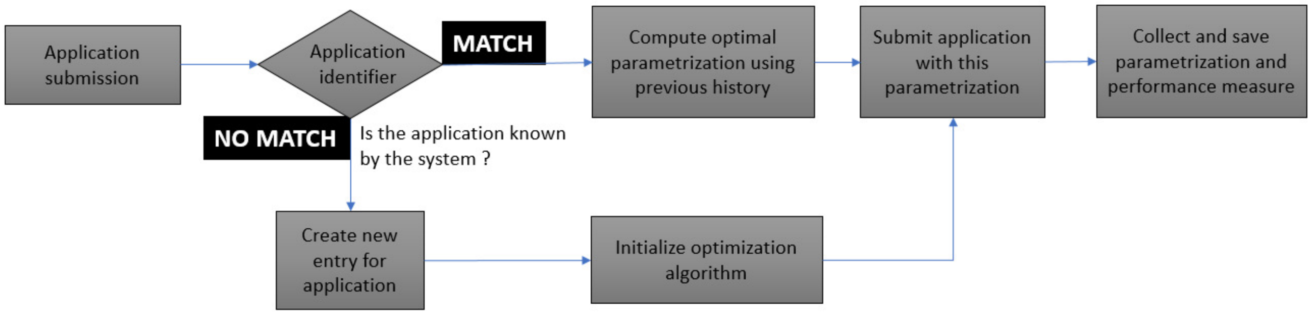

Application submission
Application identifier
MATCH
Compute optimal parametrization using previous history
Submit application with this parametrization
Collect and save parametrization and performance measure
NO MATCH
Is the application known by the system ?
Create new entry for application
Initialize optimization algorithm

FIGURE 4 The auto-tuner blocks and workflow

the previous parametrization and their corresponding output. This creates a feedback loop which continuously improves the application's execution time as shown in figure 2.

5.2 | Initialization strategy and implementation of the heuristics

We compared the three heuristics presented in section 4.3, and practically implemented the following variants:

1. Surrogate models:

   We used Gaussian Process Regression to perform the regression of the function and compared the results yielded by the two acquisition methods described in section 4.3. To compute the acquisition criterion, we simply evaluated the function over the whole grid and selected the maximum value (which is equivalent to using a brute force optimizer).

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 in any dimension of the parametric space. Two cooling schedules are tested: a multiplicative and logarithmic one [32], with a cooldown factor set to 10. 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 (or roulette wheel pick), which selects the next parametrization using a random law proportional to the fitness of each candidate, 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 [33]. Crossovers were made using single and double point crossovers when the size of the parametric space allowed it [34]. As this method also requires defining neighboring parametrizations, we opted for the same method as for simulated annealing: a parametrization can mutate into any adjacent parametrization using a random walk. We set the random Bernoulli law parameter which randomly triggers the mutation to a probability of 0.1.

The initialization of the algorithm uses Latin Hypercube Design [23].

6 | TESTS AND VALIDATION

To test, validate and compare the different heuristics variants described above to determine the most suitable one for online tuning, we performed a total of 8 experiments, using 5 different benchmarking applications, on the two accelerators described in section 3. To compute the metrics on realistic trajectories, the auto-tuning loop is simulated to confront the heuristics to real-life conditions. To do so, we build datasets which contain the execution times corresponding to an exhaustive sampling of the parametric space for a given application and accelerator. At each iteration, the dataset acts as the black-box in figure 2 and we query for the parametrization selected by the heuristic the corresponding execution time that was collected during the experiment. The optimum collected from the exhaustive sampling of the space acts as the ground truth and allows us to compare the results given by our methodology to a reference. On each of these datasets we run several variants of each heuristic and collect the optimization trajectories. To analyze them, we define several evaluation metrics to characterize the relevance of each heuristic when used for both offline and online tuning. We present and discuss these criteria in the following section.

6.1 | Evaluation criteria

Comparing the different heuristics requires the definition of metrics that can be used to rank the different heuristics according to their efficiency for performing auto-tuning of I/O accelerators in an offline and online settings.
To account for all the constraints described in section 3.3, we define the six metrics in table 3 to rank the heuristics. The first ones (S, DistOpt) evaluate the heuristics' ability to find the true optimal value within a finite budget, by measuring respectively their success rate, and the distance to the optimum found by the heuristic. While these metrics are 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 (for example in an offline setting), it is not enough for online tuning which requires consistency and stability in the results. The tuner shipped with the I/O 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.
This stability criterion is captured by two additional metrics (AvgDist, EC), designed to evaluate the stability 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 on average for computing the next parametrization to evaluate at the end of each run.

The six metrics are measured on 50 instances of each heuristic and their variants, to average any effect that could be linked to the random components of the algorithm (the initialization strategy, the selection of nearest neighbors, the number of mutations, the temperature restarts ...).

TABLE 3 Description of the metrics used to compare the different heuristics

Metric name	Code	Description	Field specific interpretation
Success	S	Whether or not the optimal parametrization was selected	Whether or not the algorithm optimally parametrized the accelerator
5% of the optimal	5%Opt	Number of iterations required to reach a parametrization which is within 5% of the optimal point	How many iterations my job requires to be instrumented to reach a good enough parametrization
Distance between optimum and correct value	DistOpt	The distance between the selected optimum and the true optimum	The difference between the optimal execution time the user could have had and the execution time returned by the selected 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 summed difference of regressions (i.e. the application performing worse than it has in the past)	The cumulative time the user sees its application performing worse than he had already seen it
Elapsed time	ET	The computation time	The time spent by the loop for computing the next solution

6.2 | Benchmarking 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, freely available as open-source software [35].

The experimentation datasets were built using the exhaustive search heuristic of the Smart HPC Application MANager, also freely available as Open-Source software [36]. We used this software to test all values of a parametric grid for different benchmark applications.

All the benchmarking applications are run using an isolated storage systems to minimize potential interferences. The experiments are run on compute nodes, which consists of an Intel Xeon E5-2670 cpu (16 physical cores, 32 virtual), bi-socket, and 62 GB of DDR4-DRAM. The datanode, i.e. the node that hosts the SBB server, is made of an Intel Xeon E5-6130 cpu (32 physical cores, 64 virtual), bi-socket, 190GB of RAM and 15TB of NVME disk space is available. The compute nodes are connected to the Data node through EDR Infinity Band (12.5GB/s max bandwith). The back-end parallel filesystem is a Lustre bay[37] of 40TB.

6.3 | Tested applications

The experiments use five I/O benchmarking applications, representative of some I/O patterns encountered in scientific HPC applications. They have been carefully taken because they have good potential for acceleration by the accelerators and sensible to the parameters. They also present very diverse parametrization landscape and this diversity will allow to compare and understand the heuristics' benefits. It also allows to focus on the part that is optimizable in this case, the I/O time of applications, and to take the execution time as the optimization target for the application. It also reduce experiment durations to collect more data, as an exhaustive grid search for a scientific HPC applications would take months to run.

Two applications are designed specifically for the Smart Burst Buffer and three for the Small Read Optimizer. Two different back-ends are tested for the Small Read Optimizer. This results in a total of a eight experiments. The number of parameters tested per experiment and its estimated duration is described in table 4. Through the remainder of the paper, we will identify an application by its *Application ID* as available in this table, and, when relevant, we will identify the used backend by suffixing it to the application ID. For instance, the dataset obtained by running the application `SRO.1` with a `NFS` backend will be designated as `SRO.1.NFS`.

TABLE 4 Summary of the experiments performed to compare the heuristics

Accelerator	Application ID	Back-end	Number of runs	Total elapsed time (s)
Smart Burst Buffer	SBB.1	Lustre	4500	316336 ($\approx$ 87 hours)
	SBB.2	Lustre	1024	76135 ($\approx$ 21 hours)
Small Read Optimizer	SRO.1	Lustre	1920	113380 ($\approx$ 31 hours)
	SRO.1	NFS	1920	116078 ($\approx$ 32 hours)
	SRO.2	Lustre	1920	40065 ($\approx$ 11 hours)
	SRO.2	NFS	1920	66676 ($\approx$ 18 hours)
	SRO.3	Lustre	1920	24886 ($\approx$ 6 hours)
	SRO.3	NFS	1920	37331 ($\approx$ 10 hours)
Total	8 experiments	2 backends	16684	790890 ($\approx$ 219 hours)

6.3.1 | SBB experiments plan

For the reasons mentioned above, we selected two benchmark applications relevant for the tuning problem of the SBB. Both experiments use the open source I/O benchmark fio [38] to generate the I/O workloads. The fio configuration of the 2 applications is summarized in table 5.

TABLE 5 Parametrization of the fio benchmark for SBB Experiments

ID	Benchmark	I/O pattern	I/O volume	Read-write distribution (%)	Block size	Number of processes
SBB.1	I/O bandwidth sensitive	Sequential	200GB	0-100	10 MB	8
SBB.2	I/O latency sensitive	Random	40 GB	75-25	4K	32

1. **SBB.1: I/O bandwidth sensitive benchmark**

   The benchmark performs sequential writes by chunks of 10 MB, for a total of 200GB of total I/O volume, 8 processes. It simulates a checkpoint being written simultaneously by all application's processes which saturates the available RDMA bandwidth. To make this application interesting for the optimizer, we set the size of the RAM and NVME caches to respectively 40GB and 150GB as well as limiting to 8 the number of usable cores on the datanode. This is a more representative run as datanodes are usually shared between jobs and taylored to force the tuner to find trade-off between the ressources to allocate to the different SBB threads pools as it runs on a limited set of cores. The optimizer must tune the SBB to avoid bottleneck on the different stages of the SBB and exploit as much as possible the available RDMA bandwidth.

2. **SBB.2: I/O latency sensitive benchmark**:

   It consists in performing random read-write I/Os, 75% reads and 25% writes, by chunk of 4k, 40GB of total I/O volume, 32 processes. This scenario is latency bound because many processes are doing small I/Os, overloading the SBB with many requests that it has to respond as quickly as possible. This is putting the pressure on the SBB front-end, the tuner must learn to give enough resources for RDMA connections (RDMA polling threads) and insertion into the RAM cache (workers threads). There are also a significant fraction of reuse on the cache, that can be exploited for performance improvement by setting the RAM cache threshold high enough. This scenario does not explore the NVME cache related options (flash destagers, flash threshold and NVME cache size) because it does not benefit from the NVME cache. Indeed, for this application, the whole I/O dataset (40GB) fits into the RAM cache and the burst buffer will not require NVME to hold the data into cache.

The tested values for each parametric dimension of the SBB parametric grid are detailed in table 6.

TABLE 6 Description of the exhaustive parametric grid for the SBB experiments

Parameter name	Minimum value	Maximum value	Step
SBB.1			
Worker Threads	1	16	power of 2
RAM destagers	1	16	power of 2
RAM cache threshold	10%	90%	20%
Flash destagers	1	16	power of 2
Flash cache threshold	10%	90%	20%
SBB.2			
RDMA polling threads	1	8	power of 2
Worker Threads	4	16	4
Ram Destagers	4	16	4
RAM cache size	10GB	80GB	10GB
RAM cache threshold	50%	90%	40%

6.3.2 | SRO experiments plan

As described in section 3, the Small Read Optimizer is designed to speed-up pseudo-random accesses made by HPC applications. These type of accesses correspond to an application exploiting a given zone of the file and then jumping to another non-adjacent zone (for example, this happens when reading a table in a non-sequential order.

Three different applications are designed based on a in-house I/O benchmark which acts by cutting the file into a certain number of conceptual chunks. The location in the file of each of these chunks is called *lead*. Around each of lead, the application performs random accesses within a certain distance of the lead, called *scatter*. Leads emulate different processes accessing the same file at the same time with different offsets, number of processes being the *number of leads*. At each operation, the operations are translated by a window of size *lead advance*. A schematic explanation of leads and scatter is provided in figure 5.

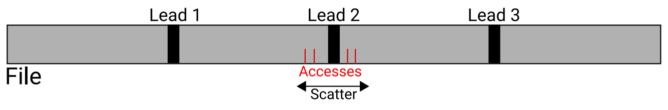


FIGURE 5 Schematic representation of leads and scatters

A summary of the parametrization of the benchmark for the three tested patterns is described in table 8.

1. **SRO.1**: 4 large hotspots (50MB of scatter) with a 10MB sliding window.
2. **SRO.2**: 220 smaller hotspots (10MB of scatter) with no sliding window.
3. **SRO.3**: 100 hotspots with a scatter for each lead picked randomly from a uniform law, ranging from a very small value (50 kB) to a very large one (500MB). The number of operations per lead is also selected randomly using a uniform law, ranging from 100 to 500 operations

The execution times of each application are recorded for 1920 distinct parametrizations. The tested values for each parametric dimension are detailed in table 7.

TABLE 7 The exhaustive parametric grid for the SRO experiments

Parameter name	Minimum value	Maximum value	Step
Cluster threshold	2	102	20
Binsize	262144	1048576	262144
Prefetch	1048576	10485760	1048576
Sequence length	50	100	750

All selected patterns presented in table 8 perform only small random read operations, as they are the ones targeted by the accelerator. Those 3 benchmarks have been chosen because they are I/O patterns found in industrial applications that we studied, they can be optimized with SRO and

TABLE 8 I/O pattern parametrization for each application for SRO experiments

ID	Number of operations	Scatter width	Number of leads	Successive operations per lead	Lead advance
SRO.1	500000	50M	4	1000	10MB
SRO.2	500000	10M	220	1	0
SRO.3	24489	$\mathcal{U}(50\text{k}, 500\text{M})$	100	$\mathcal{U}(100, 500)$	0

sensitive to its parameters. The SRO.1 is the best case scenario for SRO and the tuner must be very aggressive, lowering the threshold to prefetch as much as possible and set a prefetch size optimally to a value close to the scatter. SRO.2 has lots of leads writing at the same time, the challenge here for the tuner is focused on finding a sequence length high enough to collect enough samples to start prefetching. With variables length for scatter and I/O for SRO.3, the tuner is challenged to find a trade-off in its aggressiveness to prefetch, both in term of prefetch size and cluster threshold. This induces a strong disparity in the applications to optimize, as parameters have a different impact depending on the application and its backend. This is important as it gives many different landscapes for the different optimization heuristics, and will give results computed on very different optimization functions.

Two different back-ends, a Lustre bay of size 40TB and a NFS of size 20GB are considered for SRO experiments that results in a total of 6 datasets. The latter being much slower and smaller, prefetch strategy must adapt significantly and it changes the shape of the parametrization landscape and introduce diversity in the experiment. Experiment file sizes have been set to 100GB with a Lustre backend and 20GB with a NFS backend.

In conclusion, we provide 8 applications that present diverse optimization landscape to compare the heuristics on. These datasets are meaningful in the context of HPC applications, both in terms of I/O volume and I/O patterns.

7 | RESULTS AND DISCUSSION

For each variant of the heuristic of the auto-tuning methodology described in 5.2, and for each dataset, the metrics introduced in table 3 are computed. Each heuristic is run 50 times to average any random behaviors and each algorithm has a total budget of 100 iterations per run, the first 10 being an initialization step using a Latin Hypercube Design as introduced in section 5.2.

For each of the variant of the heuristics, the scores for the metric DistOpt which describes the quality of the optimization and AvgDist which describes the stability of the optimization, are compared and discussed. To represent the trade-off between stability and optimization quality, we consider the sum of these two variables. To gain insight on the benefits of guided search, a random sampler without replacement is used, abbreviated as RS: this sampler randomly selects a parametrization on the parametric space at each iteration.

7.1 | Comparison of the heuristics

In this section, the different heuristics are discussed and compared base on their metrics. The variety in the behavior of the heuristics attest to the variety in the optimization landscapes.

7.1.1 | Quality of the optimization

Table 9a points that Genetic Algorithms (GA) are on average the best at finding the optimal parametrization with an average distance of 3.20% to the ground truth, closely followed by Surrogate Models with 3.33% and Simulated Annealing shows less optimization power with 5.49% average distance. As can be seen in figure 6, the best optimizer between Surrogate Models and Genetic Algorithm depends on the optimized application. Surrogate Models tend to be better at optimizing SRO but Genetic Algorithms are better for SBB based datasets. This is mostly due to the smoother

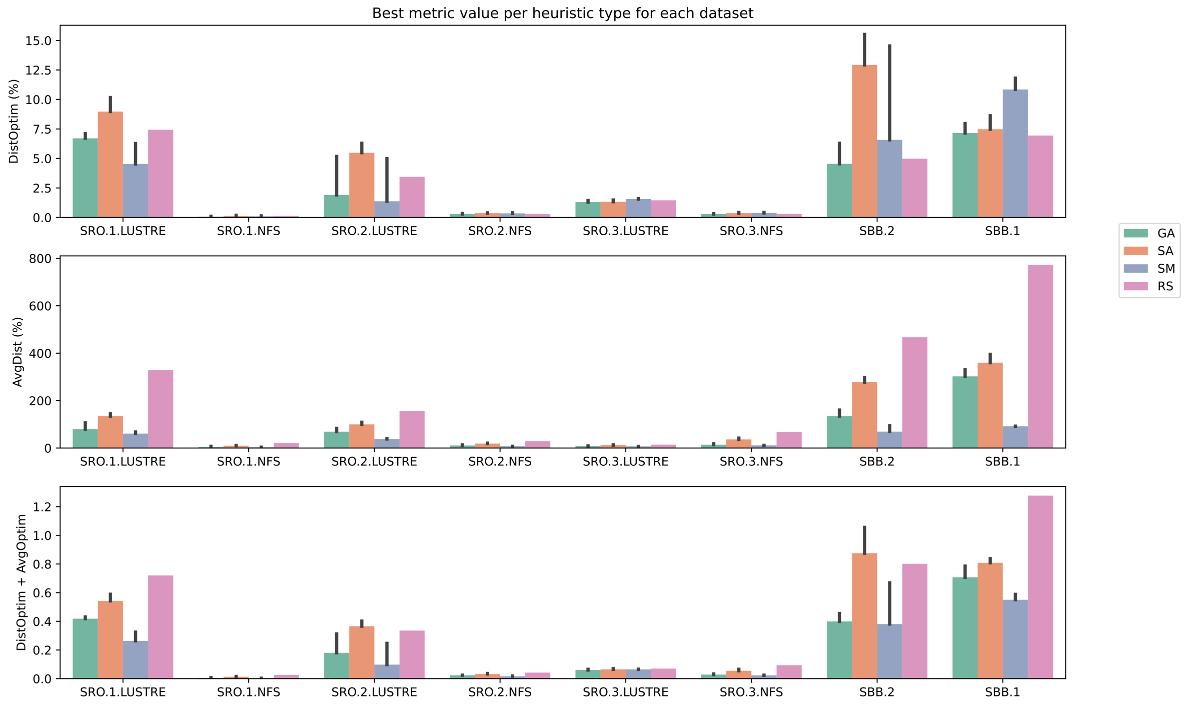


FIGURE 6 Best values for DistOptim, AvgDist and the sum of both for every heuristic

landscape of the SBB dataset and the progressive effect of the parameters on the performance of the application, as genetic algorithms are locality based rather than Surrogate Models. Because the SRO datasets have a lot more irregularities in the effect of the parameters and a less continuous landscape, Surrogate Models perform better because they are not constrained to locality as Genetic Algorithms are. Overall, Genetic Algorithms and Surrogate Models have close optimization properties, with Genetic Algorithms performing better both in mean (3.20% for Genetic Algorithms against 3.33% for Surrogate Models) but Surrogate Models performing best in median (1.73% for Genetic Algorithms and 1.48% for Surrogate Models).

The results given by using either of these two algorithms show the strong optimization and adaptation capability of the auto-tuner that we suggest, as it provides solutions close to the ground truth and the optimum for every diverse dataset.

7.1.2 | Stability of the optimization

The results concerning the stability of the optimization are reported in table 9b and in figure 6. As expected, the Random Sampler provides significantly less stability in the optimization trajectory. This shows the importance of using guided search algorithm when auto-tuning in a context

TABLE 9 Best heuristics for DistOptim and AvgDist and their corresponding metrics (aggregated over all datasets)

(a) For the best DistOptim

Heuristic	Variant	S	5%Opt	DistOpt	AvgDist	EC	ET
GA	Tournament - Single - 0.5	7.74	23.07	3.20	140.75	129.11	10.39
SM	EI	4.17	28.74	3.33	40.15	54.88	95.50
SA	No restart - Logarithmic	0.59	X	5.50	154.34	135.01	10.38

(b) For the best AvgDist

Heuristic	Variant	S	5%Opt	DistOpt	AvgDist	EC	ET
GA	Probabilistic - Double - 0.1	15.44	15.90	4.04	82.83	74.34	10.96
SM	MPI	4.17	28.74	3.33	40.15	54.88	95.50
SA	Threshold - Multiplicative	0.0	X	5.34	128.03	117.76	9.58

(c) For the best AvgDist + DistOptim

Heuristic	Variant	S	5%Opt	DistOpt	AvgDist	EC	ET	DistOptim + AvgDist
GA	Tournament - Single - 0.1	2.63	15.44	4.04	82.83	74.34	10.96	0.19
SM	EI	3.49	28.74	3.33	40.15	54.88	95.50	0.18
SA	Threshold - Multiplicative	1.19	11.28	5.03	120.38	111.07	10.43	

where stability matters and it reduces the performance of using Random Sampling for optimization. When taking solely into account the AvgDist metric, Surrogate Models perform better than Genetic Algorithms (40.14% for the former and 82.83% for the latter) when using MPI, at the cost of a reduced optimization efficiency (3.48% for the former and 2.62% for the former). They both outperform Simulated Annealing in stability (128.02%) and distance to the optimum (5.33%). The stability of Surrogate Models is due to the fact that they are not constrained by locality and can switch to different zones of the landscapes as soon as the probability of the execution time getting worse.

When considering each dataset separately, some disparities are found in heuristics' behavior. Indeed, Surrogate Models offer the most stable behavior for every dataset but the SBB.2 one where it performs significantly worse than the Genetic Algorithms.

7.1.3 | Considering both properties

Considering the sum of the normalized variables *DistOptim* and *AvgDist* in table 9c allows to rank the heuristics according to both their optimization quality and their stability. Surrogate Models provide the best value for this score (0.18), closely followed by Genetic Algorithms (0.19) and Simulated Annealing (0.36).

In conclusion, Surrogate Models, parametrized using Expected Improvement gives us some strong convergence properties as well as some guarantees in stability. If considering only these two properties, Surrogate Models are the best candidate for tuning the accelerators.

7.1.4 | Discussion of other metrics

When looking at the elapsed time (metric EC), Surrogate Models take more time to compute the next parametrization (around 95 seconds) compared to Genetic Algorithms and Simulated Annealing (less than 10 seconds). This is completely aligned with the complexity of the algorithms.They also consume more resources.

When looking at table 9, success rates stay very low (below 5% for the metric IS) for all the heuristics, even though good results are found in terms of distance to the optimum. This attests to both the complexity of the landscape to optimize and the presence of local optimums really close to the global optimum. As the heuristic attains this local optimum, it does not look further for better solutions and remains in the vicinity of this local optimum. To still have a sense of convergence and how fast the user can expect to have an efficient parametrization of the accelerator, we focus on the convergence rate as the number of iterations required to come as close as 5% to the ground truth optimum.

To be at 5% of the optimum, Genetic Algorithms require 34 steps while Surrogate Models have a slower convergence with 38 steps. Simulated Annealing has a slower convergence rate with 87 required steps to reach convergence, which is almost the entire budget. The faster convergence rate of Genetic Algorithms make them very important in our context of HPC applications, as HPC applications can run for long hours and even only 4 steps can make a strong difference.

7.1.5 | Results consistency

As each of the different variant of the heuristics is repeated 50 times, we must evaluate whether or not there is a strong difference in terms of results and optimum found at each new launch of the heuristic. This allows to evaluate how much of the found optimization strength can be attributed to the random components of the heuristics. The consistency of results of each heuristic are evaluated by looking at the statistical distribution of the DistOptim and the AvgDist metrics in figure 7, in terms of distance to the optimum and mean loss.

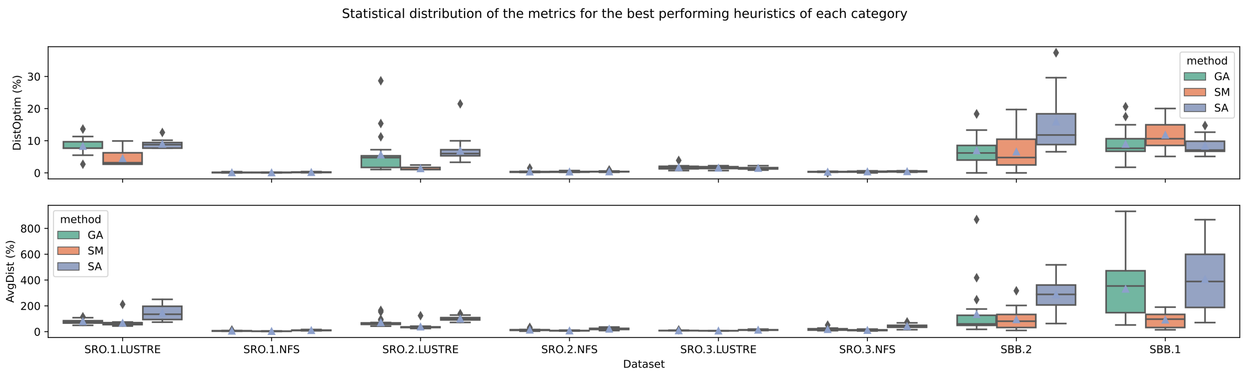


FIGURE 7 Statistical distributions of metrics over the selected heuristics

It shows a non negligable impact of randomness on the results, both when measuring the distance to the optimum and the mean loss, even in the case of purely deterministic algorithms like surrogate models. This shows the importance and the impact of the initialization plan on the performance of the algorithm. The SBB datasets are more sensitive to this random factor, because they have a strong disparity and many outliers. A bad initial parametrization, chosen among outliers, has a strong negative impact on the rest of the behavior of the heuristic, especially when the algorithm is location based.

7.2 | Impact of the hyperparameters on the heuristics' behavior

The different hyperparameters used to setup the heuristics have an impact on their behavior. Figures 8 describes the impact of the hyperparameters on genetic algorithms. The impact of the selection method depends on the accelerators and thus the size of the parametric space. Probabilistic pick provides better optimization results when the parametric space is smaller as with SRO accelerator. In terms of stability, the selection method seems to have no importance. The choice of the crossover method has an impact on the quality of the optimization as double crossovers are more efficient and increase stability for every dataset.

The mutation rate is the one with the most impact on the heuristics behavior, as can be expected from the definition of the algorithm. The higher the mutation rate, the better the optimization quality because the parametric space is explored quicker but also the less stable the optimization is because of the higher frequency of changes.

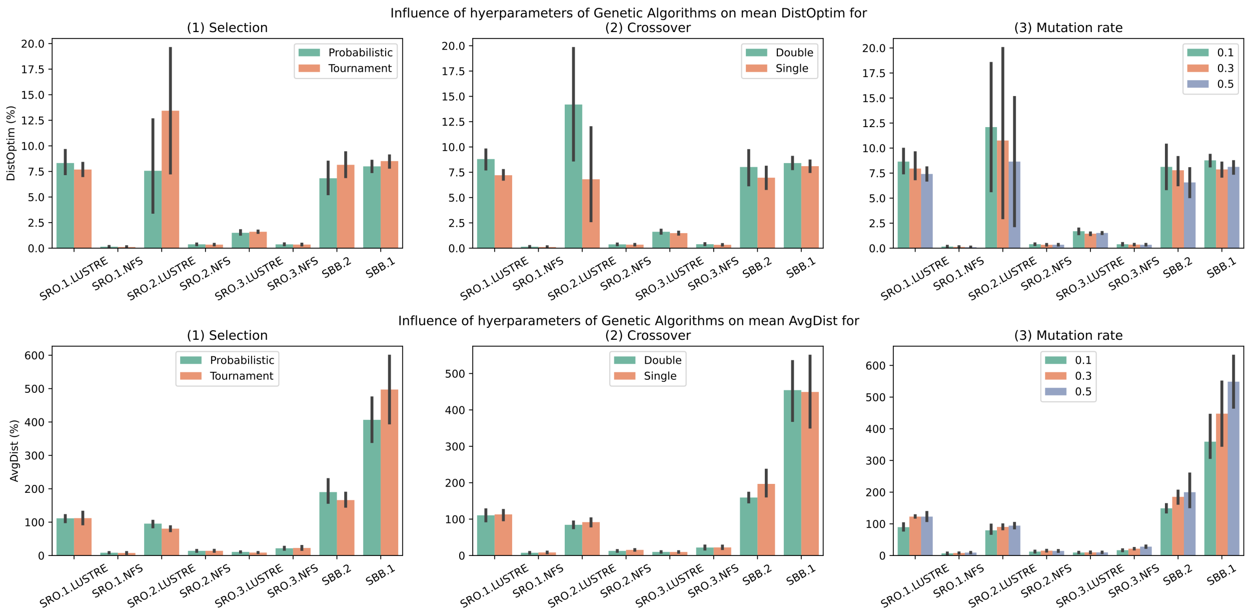


FIGURE 8 Impact of hyperparameters on quality and stability of optimization for Genetic Algorithms

In the case of Simulated Annealing, shown in figure 9, the cooling schedule has almost no impact on the behavior of the heuristic. The restart schedule has an impact on the quality of the optimization and not providing any restart improves the quality of the optimization, probably because restarting the optimization is too expensive in a limited budget setting.

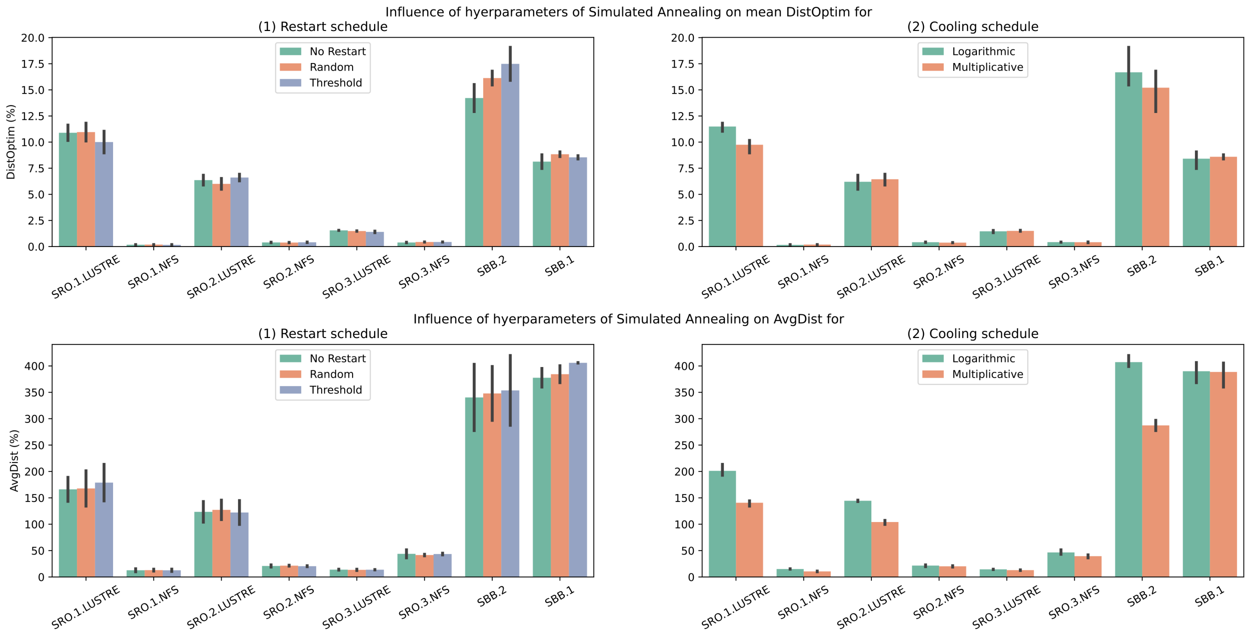


FIGURE 9 Impact of hyperparameters on quality and stability of optimization for Simulated Annealing

For surrogate models, the results in figure 10 show that using the Expected Improvement function improves the quality of the optimization, but also reduces its stability.

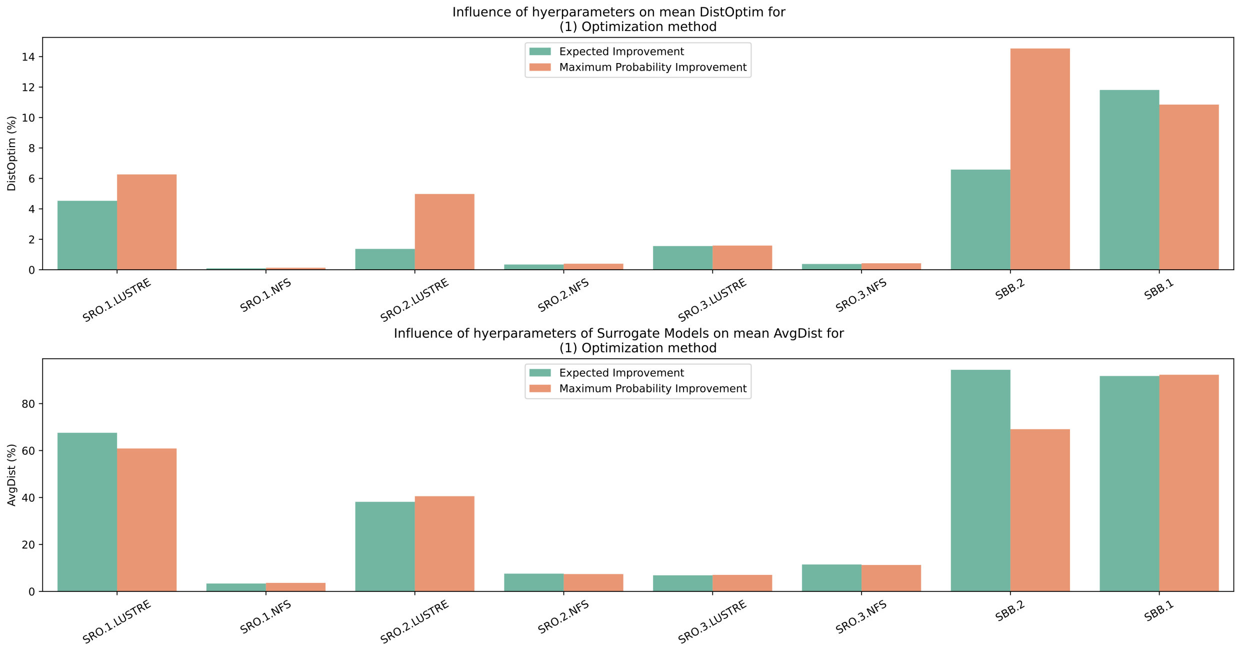


FIGURE 10 Impact of hyperparameters on quality and stability of optimization for Surrogate Models

8 | CONCLUSION AND FUTURE WORKS

In this work, we have successfully demonstrated the usefulness of using black-box optimization for auto-tuning both a pure software and a mixed soft-hardware I/O accelerator when used with I/O intensive applications. We have considered three heuristics with different variants in order to choose the most suitable one to include in our online tuner. In order to estimate their suitability, we have ranked them according to 6 defined metrics which describe their behavior in terms of both optimization quality and stability of the trajectory. A random sampler has been selected to act as a reference truth for comparing the heuristics. Their relevance for both online and offline tuning has thus been evaluated.

For both I/O accelerators, surrogate models using Gaussian Process regression and Expected Improvement as an acquisition function offer the best trade-off between optimization quality and stability of the trajectory, and outperforms a random sampler. With a distance to the true minimum inferior to 4% for every application, our auto-tuner exhibits good convergence properties. We have also proved its transparency by showing its evaluation cost and average distance to the minimum to be within an acceptable range for the user. We have also shown its universality by applying the algorithms to two I/O accelerators that operate very differently. As we have found convergence rate inferior to 40 steps for reaching 5% of the optimal value, we have also demonstrated that the auto-tuner can operate in a sparse production environment.

To improve upon this work, we plan to investigate the robustness of black-box optimization algorithms to the stochastic context. For now, we have operated in a deterministic context, while HPC clusters running in production can have a stochastic behavior. Indeed, when resources are shared among users, 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 I/O behavior and workload, and transfer the convergence information to the auto-tuning process.

References

[1] M. Nabe L. Vincent and G. Goret. Self-optimization strategy for io accelerator parameterization. In International Conference on High Performance Computing, pages 157–170, 2018.

[2] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming parallel i/o complexity with auto-tuning. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pages 68:1–68:12, 2013.

[3] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. W. Vuduc. Autotuning in high-performance computing applications. Proceedings of the IEEE, 106:2068–2083, 2018.

[4] S. Robert, S. Zertal, and G. Goret. Auto-tuning of io accelerators using black-box optimization. In Proceedings of the International Conference on High Performance Computing Simulation (HPCS), 2019.

[5] M. Gendreau J-Y. Potvin. Handbook of metaheuristics. International series in Operations Research and Management Science, 2010.

[6] D. Desani, V. Gil Costa, C. A. C. Marcondes, and H. Senger. Black-box optimization of hadoop parameters using derivative-free optimization. In 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pages 43–50, 2016.

[7] P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. CoRR, abs/1606.06543, 2016.

[8] V. Dalibard, M. Schaarschmidt, and E. Yoneki. Boat: Building auto-tuners with structured bayesian optimization. In Proceedings of the 26th International Conference on World Wide Web (WWW'17), pages 479–488, 2017.

[9] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. Autotuning in high-performance computing applications. Proceedings of the IEEE, 106(11):2068–2083, 2018.

[10] K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In 2008 IEEE International Conference on Cluster Computing, pages 421–429, 2008.

[11] P. Knijnenburg, T. Kisuki, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. The Journal of Supercomputing, 24:43–67, 01 2003.

[12] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. In 2012 Innovative Parallel Computing (InPar), pages 1–9, 2012.

[13] T. Miyazaki, I. Sato, and N. Shimizu. Bayesian optimization of hpc systems for energy efficiency. In High Performance Computing, pages 44–62. Springer International Publishing, 2018.

[14] Z. Cao. A Practical , Real-Time Auto-Tuning Framework for Storage Systems. PhD thesis, State University of New York at Stony Brook, 2018.

[15] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E Long. Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, pages 42:1–42:14, 2017.

[16] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18, pages 893–907, 2018.

[17] B. Behzad, S. Byna, M. Prabhat, and M. Snir. Pattern-driven parallel i/o tuning. In Proceedings of the 10th Parallel Data Storage Workshop, pages 43–48, 11 2015.

[18] G. Aupy, O. Beaumont, and L. Eyraud-Dubois. What Size Should your Buffers to Disks be? In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2018.

[19] G. Aupy, O. Beaumont, and L. Eyraud-Dubois. Sizing and Partitioning Strategies for Burst-Buffers to Reduce IO Contention. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019.

[20] Tools to improve your efficiency. https://atos.net/wp-content/uploads/2018/07/CT$_J$1103$_1$80616$_R$Y$_{FT}$OOLSTOIMPR$_W$EB.pdf.[Online; accessed2021/03

[21] Atos. Atos steps up with nvmeof flash accelerator solutions. https://atos.net/en-na/2019/news-na_2019_02_22/atos-steps-nvmeof-flash-accelerator-solutions, 2019.

[22] M. Jette, A. Yoo, and M. Grondona. Slurm: Simple linux utility for resource management. 07 2003.

[23] K. T. Fang, R. Li, and A. Sudjianto. Design and Modeling for Computer Experiments (Computer Science & Data Analysis). Chapman & Hall/CRC, 2005.

[24] Y. Hamadi V. K. Ky, C. D'Ambrosio and L. Liberti. Surrogate-based methods for black-box optimization. International Transactions in Operational Research, (24), 2016.

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

[26] D. Dunlop, S. Varrette, and P. Bouvry. On the use of a genetic algorithm in high performance computer benchmark tuning. In 2008 International Symposium on Performance Evaluation of Computer and Telecommunication Systems, pages 105–113, 2008.

[27] Valentin Plugaru, Fotis Georgatos, Sébastien Varrette, and Pascal Bouvry. Performance tuning of applications for hpc systems employing simulated annealing optimization. Technical Report CSC-10993/10301, University of Luxembourg, 09 2013.

[28] Holger H. Hoos Frank Hutter and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In International Conference on Learning and Intelligence Optimization (LION'11), pages 507–523, 2011.

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

[30] C. D. Gelatt S. Kirkpatrick and M. P. Vecchi. Optimization by Simulated Annealing, volume 220. Science, 1983.

[31] L. Davis. Handbook of Genetic Algorithms. Van Nostrand Reinhold, 1991.

[32] Y. Nourani and B. Andresen. A comparison of simulated annealing cooling strategies. Journal of Physics A: Mathematical and General, 31(41):8373–8385, oct 1998.

[33] N. Saini. Review of selection methods in genetic algorithms. International Journal of Engineering and Computer Science, 6(12):22261–22263, 2017.

[34] N. Saini. Review of selection methods in genetic algorithms. International Journal Of Engineering And Computer Science, 3, 2017.

[35] https://github.com/bds-ailab/shaman. [Online; accessed 2021/03/10].

[36] Sophie Robert, Soraya Zertal, and Gaël Goret. Shaman: An intelligent framework for hpc auto-tuning of i/o accelerators. In Proceedings of the 13th International Conference on Intelligent Systems: Theories and Applications, SITA'20, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Lustre filesystem. http://lustre.org/. [Online; accessed 2021/03/10].

[38] Fio benchmark. https://fio.readthedocs.io/. [Online; accessed 2021/03/10].

How to cite this article: S. Robert, S. Zertal, G. Vaumourin, and P. Couvée (2021), CCPE, *Computation and concurrency:Practice and Experience* <2021> <vol> Page <xxx>-<xxx>