SHAMan: an intelligent framework for HPC auto-tuning of I/O accelerators

Sophie Robert
Li-PaRaD
University of Versailles
Versailles, France
Atos BDS R&D Data Management
Echirolles, 38130, France
sophie.robert@atos.net

Soraya Zertal
Li-PaRaD
University of Versailles
Versailles, 78000, France

Gaël Goret
Atos BDS R&D Data Management
Echirolles, 38130, France

Abstract

Like most modern computer systems, High Performance Computing (HPC) machines integrate many highly configurable hardware devices and software components. Finding their optimal parametrization is a complex task, as the size of the parametric space and the non-linear behavior of HPC systems make hand tuning, theoretical modeling or exhaustive sampling unsuitable for most cases. Auto-tuning methods relying on black-box optimization have emerged as a promising solution for finding systems' best parametrization without making any assumption on their behaviors. In this paper, we present the architecture of an auto-tuning framework, called **S**mart **H**PC **A**pplication **MAN**ager (SHAMan), that integrates black-box optimization heuristics to find the optimal parametrization of an Input/Output (I/O) accelerator for a HPC application. We describe the conceptual and technical architecture of the framework and its native support for HPC clusters' ecosystem. We detail in depth the stand-alone optimization engine and its integration as a service provided by a Web application. We deployed and tested the framework by tuning an I/O accelerator developed by the Atos company on a HPC cluster running in production. The tuner's performance is evaluated by optimizing 90 different I/O oriented applications. We show a median improvement of 29% in speed-up compared to the default parametrization and this improvement goes up to 98% for a certain class of applications.

CCS Concepts • **Theory of computation → Optimization with randomized search heuristics**; **Online learning algorithms**; • **General and reference → Performance**.

Keywords Auto-tuning; Input/Output; I/O accelerators; High Performance Computing; Performance optimization; Randomized search heuristics

ACM Reference Format:
Sophie Robert, Soraya Zertal, and Gaël Goret. 2020. SHAMan: an intelligent framework for HPC auto-tuning of I/O accelerators. In *13th International Conference on Intelligent Systems: Theories and Applications (SITA'20), September 23–24, 2020, Rabat, Morocco.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3419604.3419775

SITA'20, September 23–24, 2020, Rabat, Morocco

ACM ISBN 978-1-4503-7733-1/20/09…$15.00
https://doi.org/10.1145/3419604.3419775

1 Introduction

In order to make the most of their resources and improve their performance, intelligent software components and hardware devices should exhibit some self-adaptive properties and be able to automatically select the most appropriate parameters for their current workload. This problematic is particularly important in the field of High Performance Computing (HPC). As supercomputers perform a wide range of computationally intensive tasks and handle heavy I/O workloads, their different components must be configured with the most appropriate parametrization for the currently running application. 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.

We propose in this paper the architecture of a framework, called **S**mart **H**PC **A**pplication **MAN**ager (SHAMan), for finding the optimal parametrization of HPC components using black-box optimization. This framework is composed of a Web application, a command line interface, an optimization engine and a persistent storage. It takes into accounts the constraints of the HPC ecosystem and is fully compatible with workload managers. We describe a practical use-case of this framework by tuning an I/O accelerator developed by the Atos company for a given HPC application.

The main contributions of this work are:

- The description of the conceptual and technical architecture of an auto-tuning framework fully integrated with the HPC ecosystem that could be extended to wider use-cases;
- The original use-case of using black-box optimization algorithms to find the optimal parametrization of I/O accelerators;
- The evaluation of the framework on a HPC cluster running in production.

In the remainder of the paper, section 2 introduces the theoretical grounds of black-box optimization and describes three state-of-the-art heuristics shipped with SHAMan. Section 3 presents different works related to the optimization of I/O accelerators and more generally to the auto-tuning of computer systems and the already available tuning frameworks. Details about our particular use-case and I/O accelerators are given in section 4. In section 5, we present the auto-tuning framework and its architecture. Section 6 describes the conducted tests and their results. Finally, we conclude our paper and give some hints for future works in section 7.

2 Theoretical background

Systems' optimization is a well studied field and black-box optimization has emerged as a state-of-the-art solution for optimizing

non-linear systems. This section presents the general principles of black-box optimization and the three heuristics that have been implemented in the framework's optimization engine.

2.1 What is black-box optimization?

Black-box optimization refers to the optimization of a function f with unknown properties in a minimum of evaluations, without making any assumption on the function. The only available information is the history of the black-box function, which consists in the previously evaluated parameters and their corresponding objective value. Given a budget of n iterations, the problem can be transcribed as:

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

- f the function to optimize
- $\mathbb{P}$ the parameter space
- ϵ a convergence criterion between the found and the estimated minimum

Every black-box optimization process starts with the selection of the initial parameters for the algorithm. An acceptable initialization starting plan should respect two properties [11] [20] [14]: 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-collapsible property specifies that no parametrization can have the same value on any dimension. A design plan respecting this constraint is called a *Latin Hypercube Design* (LHD) [11]. The next step consists in a feedback loop, which iteratively selects a parametrization, evaluates the black-box function at this point and selects accordingly the next data point to evaluate. The higher procedure for searching an optimal solution in a parametric space is called an optimization *heuristic*.

2.2 Heuristics available in the framework

There are many black-box heuristics, and we have implemented the following set in our optimization engine because of their simplicity of implementation and their proven efficiency for HPC systems' tuning [18].

1. Surrogate models:
   Surrogate modeling consists in using a regression or interpolation technique over the currently known surface to build a computationally cheap approximation of the black-box function and to then select the most promising data point in terms of performance on this surrogate function [20] by using an acquisition function. Any regression or interpolation function can be used to build the surrogate model. Several acquisition functions can be found in the literature [20]. The most popular are the *Maximum Probability of Improvement* and the *Expected Improvement* which model each data point of the surrogate as a probabilistic variable and encodes its likelihood of being a better parametrization than the current best one [20].
2. Simulated annealing:
   Introduced in [19], the simulated annealing heuristic is a hill-climbing algorithm which can probabilistically accept a solution worse than the current one. The algorithm is started using an initial "temperature" value, which will decrease over time. Then, at each iteration, the algorithm randomly selects a parametrization neighboring the current one and computes its execution time. If the new parametrization is better than the current parametrization, it is automatically accepted. If not, a probability of acceptance is computed as a function of the system's temperature. The temperature of the system then decreases, according to a cooling schedule which ensures that the probability of accepting a solution worse than the current one lowers at each iteration until it draws to zero at the end of the algorithm.
3. Genetic algorithms:
   Genetic algorithms consist in selecting a subset of parameters, among the already tested parametrizations, according to a selection mechanism that considers the objective value of each parametrization [10]. These selected parameters are then combined to create new ones, most commonly by cutting them into n parts and alternatively concatenating them. These newly created combinations can randomly undergo a mutation by being slightly altered which ensures some randomness and diversity in the optimization process.

3 Related works

Black-box optimization has been used in several domains over the last years and has been implemented in different generic optimization tools. It has been particularly helpful in computer science for finding optimal configurations of various software and hardware systems. Within the HPC community, it has been used to improve the portability of applications across architectures using random-based heuristic searches in [5]. Surrogate modeling has also successfully been used to reduce clusters' energy consumption in [17]. The I/O community has also successfully tuned highly configurable storage systems for particular workloads by using black-box optimization tuners [8] [16] [9] [7] [6]. Reinforcement learning has also proven to efficient as an auto-tuner to optimize the performance of the Lustre filesystem in data center storage systems [16].
When it comes to auto-tuning frameworks, several have been proposed recently. Frameworks like *Google Vizier* [12] provide black-box optimization as a service. Several innovative frameworks, like Optuna [3] and Autotune [15], have been developed to find the optimal parametrization of Machine Learning models. The OpenTuner [4] framework has been developed for building domain-specific multi-objective program autotuners. However, to our knowledge, no framework specific to HPC clusters' running in production has been suggested.

4 Main use-case of the study: building an auto-tuner for I/O accelerators

The main motivation for the SHAMan framework is the development of a Web application that automatically finds the optimal parametrization of I/O accelerators. I/O accelerators are software or hardware components developed 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 backend and can make the application wait while it performs its I/O. To mitigate this problem, several I/O accelerators have been developed over the years. We focus on one of them, a pure software optimizer for data memory loads, developed by the Atos company and called the *Small Read Optimizer* (SRO). This accelerator comes with 4 parameters and its effect on the application's I/O patterns

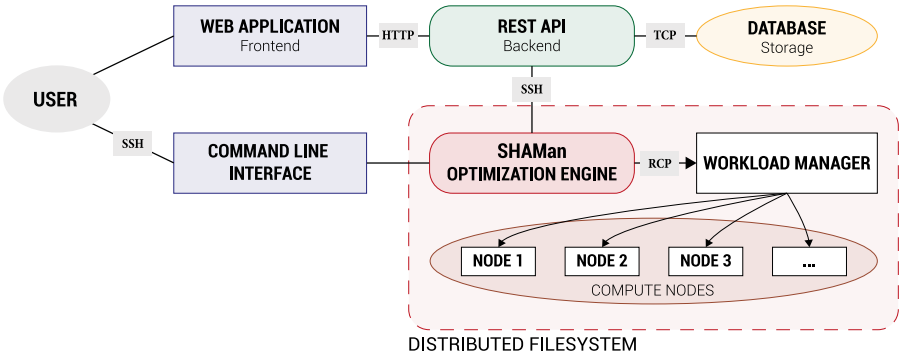

Figure 1. General architecture of the tuning framework

is highly sensitive to the parameters' values. The final version of this auto-tuning framework is to be integrated within Atos' I/O ecosystem, called the Smart Data Management suite, by allowing the user to select and tune an application through the Atos' Web interface.

5 Framework description

5.1 Terminology

Throughout this section, we will use the following terms:

- *Component*: the component which optimum parameters must be found. In our current implementation, we consider I/O accelerators.
- *Target value*: the measurement that needs to be optimized. In our current implementation, we consider the execution time of the application.
- *Application*: a program that can be run on the clusters' nodes using a workload manager and on which the target value can be measured. In our current implementation, an application is a shell file that can be submitted through the Slurm [13] workload manager and its running time is obtained using the Linux `time` command.
- *Budget*: the maximum number of evaluations the optimization algorithm can make to find the optimum value.
- *Experiment*: A combination of a component, a target value, an application and a budget that will output the best parametrization for the application and the component. We define *launching the experiment* as the start of the optimization process. An experiment is finished once the iteration budget has been spent.

5.2 Design goals and main features

The architecture is designed to respect these constraints:

- *Web user interface*: the optimization engine must be accessible through a Web interface that enables to launch new experiments and visualize previous ones;
- *No preliminary knowledge*: using the framework must not require any knowledge about the application, the component or the inner workings of the optimization process;
- *Easy to extend*: Implementing a new heuristic for the optimization engine and using it for tuning must be an easy process;
- *Integrated within the HPC ecosystem*: the framework itself must run on isolated service nodes but the tuned application must run on the compute nodes of the cluster;
- *Flexible for a wide range of use-cases*: the current implementation is specific to I/O accelerators, but the framework must be easy to adapt to any other component that would benefit from tuning.

The main features of the SHAMan framework are the possibilities to:

- Design and launch an experiment (as defined above) through a Web interface or through a command line interface;
- Visualize information about finished or running experiments, such as:
  - The optimization trajectory and the execution time associated with each tested parametrization;
  - The metadata of the experiment, such as its name, its elapsed time, its settings, *etc.*
- Store the experiment data in a permanent storage.

5.3 User workflow

There are 3 different ways to use the framework. The quickest and easiest way to use SHAMan is to use the Web interface, which enables to design and launch new experiments by selecting an application, an accelerator, and a budget. It also provides a dashboard to visualize finished experiments. SHAMan can also be accessed through a Command Line Interface to design and launch the experiments. Finally, the user can directly use the optimization engine through an interactive Python session or use it as a third-party library.

5.4 Main modules

SHAMan's main architecture and modules can be seen in figure 1. Its core is composed of:

- An analytics node which runs the optimization engine
- A vizualisation node which runs the front-end
- A back-end node which runs a REST API
- A storage node which hosts the database that contain the results of the experiments

5.5 Optimization engine

The optimization engine is a stand-alone Python library that can be used either interactively with RAM-only results or through a command-line and REST API which enables persistent storage. It

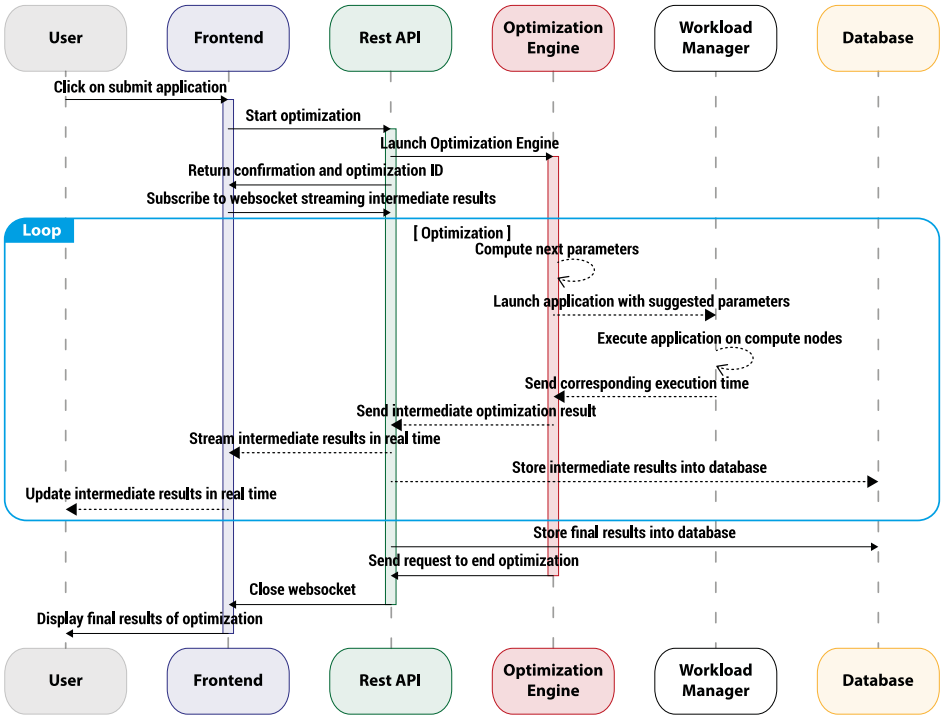

Figure 2. Application's workflow upon experiment submission

is installed on a node, separate from the compute nodes, so that it does not interfere with the running application.

The engine is separated into two distinct modules:

- A generic black-box library which comes out of the box with the framework
- A wrapper which transforms the component into a tunable black-box specific to the user's use-case

5.5.1 The black-box optimizer

The black-box optimizer module performs the black-box optimization process. Its main functionality is to optimize on a specified grid any Python object which satisfies the requirement of having a `.compute` method that takes as input a vector corresponding to a parametrization and outputs a scalar corresponding to the target value. It integrates the three heuristics described in section 2. While any third party library which performs black-box optimization could be used, we choose to code our own optimizer for the simplicity of future extensions and HPC specific support.

5.5.2 The black-box wrapper

The black-box wrapper module transforms the component undergoing tuning so that it can be considered as a black-box by the optimizer module. It must be adapted to each specific use-case. Its development consists in developing a Python object that enables the manipulation of the component by building a `.compute` method that takes as input the parameters of the component and outputs the corresponding target value. To use the framework for a custom component, the user must develop his/her own wrapper.

In the case of I/O accelerators, natively supported by SHAMan, the corresponding `.compute` method is a method which takes as input the parameters of the accelerator to tune. It then launches the application along with the accelerator with the right parametrization (by setting up the correct environment variables) through the workload manager. The application execution time is recorded by adding a `time` command and parsing its value in the output file of the workload manager.

5.6 Integration of the optimizer within a Web application

As described in figure 1, the optimization engine is integrated within a Web application to allow the user to launch and visualize experiments. A REST API ensures the communication between the modules. The main event is the submission of a new experiment and its general workflow is described in figure 2.

5.7 Implementation choices

The optimization engine is entirely written in Python. The practical implementation of the framework uses the workload manager *Slurm*[13] and the corresponding Atos' plugin for launching the SRO accelerator. The REST API is implemented with the lightweight Web micro-framework *Flask*[1]. The visualization front-end simply uses *Javascript* and the storage system uses *MongoDB*[2] database system.

6 Evaluation on practical applications

To confirm the efficiency of black-box optimization that was found in[18] and to test the tuning process on a cluster running in production, we evaluate the tuner on 90 applications to look for their optimal parametrization of the SRO I/O accelerator described in section 4.

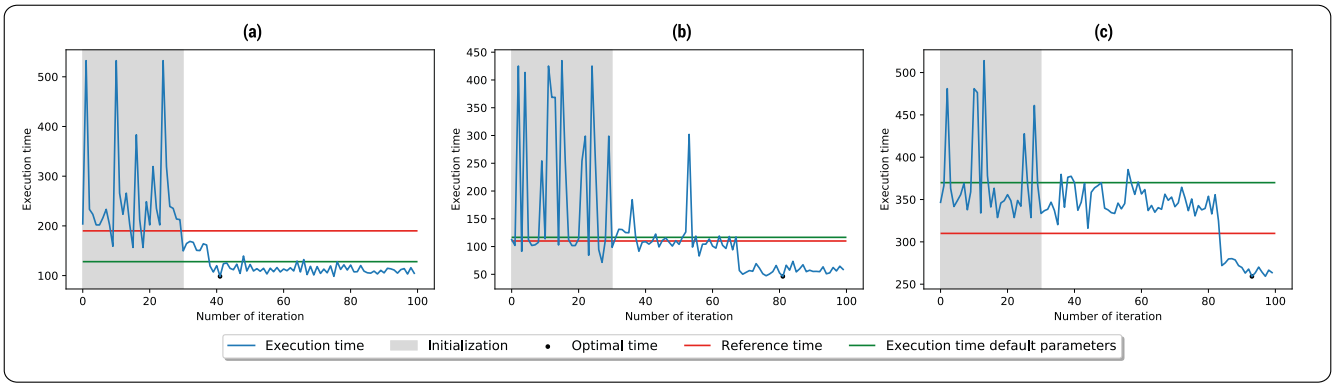

Figure 3. Three different types of optimization trajectory

6.1 Experiment design

6.1.1 Application design

90 synthetic applications are designed to test the tuning process. In order to capture different sensitivity to the accelerator's parameters, 9 categories are designed, with different I/O characteristics: different read-write ratios (0.5, 0.7, 0.9) and sizes of operation (small, medium, large). To ensure diversity for the 10 applications within each category, the other characteristics of the applications (the number of operations, the length of the file covered by each I/O ...) are drawn from random laws. Each category is labeled using the naming convention `readratio_size`. For example, the category performing small operations with a ratio of apparition of 0.5 read - write is denoted as `read05_small`).

6.1.2 Experiment layout

The tuning experiment consists in running SHAMan with a budget of 100 iterations, the first 30 being reserved for initialization and the remaining 70 for black-box optimization using genetic algorithms. This results in a total of 900 runs. Overall, the whole experiment lasted approximately 29 days. The data associated with each run was stored in the SHAMan database for further exploitation.

6.1.3 Parametric space description

The parametric grid required by SHAMan to perform the optimization upon is described in table 1. It is based on the 4 discrete parameters of `SRO` accelerator with selected values from previous works and commonly encountered I/O patterns in HPC applications.

Parameter name	Lower limit	Upper limit	Step
Cluster threshold	1	15	1
Binsize	1048576	20971520	524288
Sequence length	50	200	20
Prefetch size	2097152	41943040	1048576

Table 1. Description of the parametric grid

This amounts to a total of 606480 possible parametrizations that the black-box optimization algorithm can choose from.

6.1.4 Hardware configuration

All of the nodes described in figure 1 (including the compute nodes) were exclusively reserved for running our applications. They all have the same hardware characteristics, with 16 cores Intel Xeon, 62GB of RAM and run RHEL 7.6. To provide results that are representative of a HPC cluster running in production, the filesystem was shared by all users of the cluster.

6.2 Observed results

Throughout this section, the term *speedup* refers to the ratio of the difference in elapsed-time between the accelerated and non-accelerated run. Expressed in percentage, it quantifies the ability of the accelerator and its parametrization to speed up the application.

We use the metrics shown in figure 3, to compare the applications and the performance of the tuner:

- The **reference** time (red line) corresponds to the time spent by the application when not using any accelerator.
- The **default** time (green line) corresponds to the time spent by the application when using the accelerator with its default shipped parametrization (*Cluster threshold=2*, *Binsize=1048576*, *Sequence Length=100*, *Prefetch size=20971520*)

6.2.1 Comparison to default parametrization

When comparing the speedup corresponding to the default parametrization and the optimum parametrization found by SHAMan, we see an improvement for every category of application, as depicted in figure 4. Over all categories, the default parametrization yields a median speedup of 1.91%. When using the SHAMan framework, the speedup improves to 31.09%. The best category in terms of improvement in median speedup is `read09_large` with an improvement of 97.83%.

The categories `read09_large`, `read05_large`, `read05_small` and `read07_medium`, which were initially slowed down (in median) by the accelerator using default parametrization but sped-up through SHAMan, show that finding the right parametrization is crucial to make the most of the I/O accelerators' benefits.

6.2.2 Convergence speed

In terms of convergence speed, in median, 61 iterations are required to find the optimum over all applications. As seen in figure 5, the number of iterations required to find the optimal parametrization does not vary a lot across the different category of experiments. Reducing this relatively high number of iterations will be among our next research goals.

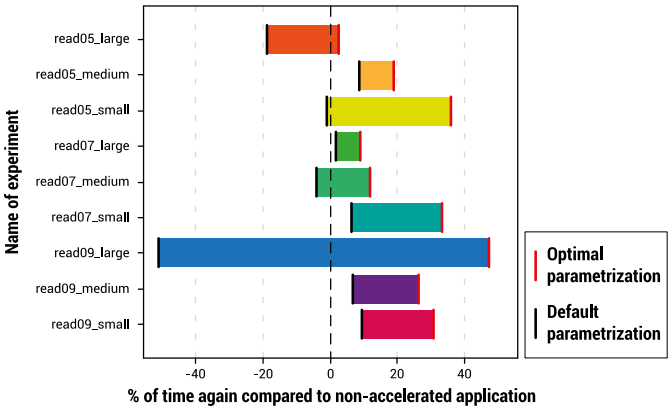

Figure 4. Comparison of default parametrization and SHAMan speedup per experiment class

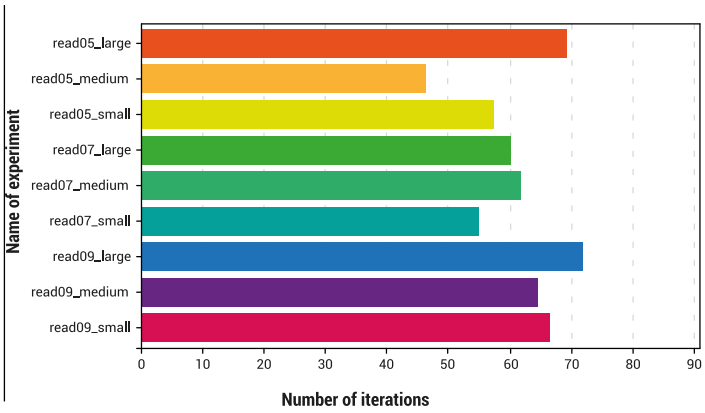

Figure 5. Median number of iterations to find optimum parametrization per experiment class

6.2.3 Example of trajectories

Figure 3 illustrates different types of trajectories that have been observed during the optimization process. In (a), the tuner brings little benefit as the optimum parametrization is close to the default parameters. However, (b) and (c) strongly benefit from the tuning process. When using default parameters, it seems that the applications should not be run with the accelerator, as it has no effect in the case of (b) and slows down the application in the case of (c). However, after tuning the application, we find that the right parameters provide a very strong speedup: the tuner reveals the usefulness of the SRO accelerator.

7 Conclusion and further works

In conclusion, we propose an auto-tuning framework for finding the optimal parametrization of HPC components. When tested on applications running on a real cluster, we show a median improvement of 29% in speed-up compared to the default parametrization and this improvement goes up to 98% for a certain class of application. It will be included within the Atos Smart I/O suite and will be launched in production.

In the near future, we will extend our framework to add a noise reduction step, in order to deal with the natural fluctuations of the cluster. We will also add the support for other I/O accelerators developed by Atos. As a long term research, we intend to build a Machine Learning model to directly infer the best parameters from the I/O profile and skip the black-box optimization process.

References

[1] [n.d.]. Flask, a lightweight WSGI web application framework. https://www.palletsprojects.com/p/flask/. Online; accessed: 2020-02-06.

[2] [n.d.]. MongoDB, the most popular database for modern apps. https://www.mongodb.com/. Online; accessed: 2020-02-06.

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. arXiv:cs.LG/1907.10902

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. Association for Computing Machinery, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[5] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (2018), 2068–2083.

[6] B. Behzad, S. Byna, M. Prabhat, and M. Snir. 2015. Pattern-driven parallel I/O tuning. In *Proceedings of the 10th Parallel Data Storage Workshop*. 43–48.

[7] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. 2013. Taming Parallel I/O Complexity with Auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. Article 68, 12 pages.

[8] Z. Cao. 2018. A Practical , Real-Time Auto-Tuning Framework for Storage Systems.

[9] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok. 2018. 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)*. 893–907.

[10] L. Davis. 1991. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.

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

[12] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley (Eds.). 2017. *Google Vizier: A Service for Black-Box Optimization*. http://www.kdd.org/kdd2017/papers/view/google-vizier-a-service-for-black-box-optimization

[13] M. Jette, A. Yoo, and M. Grondona. 2003. SLURM: Simple linux utility for resource management. *Lecture notes in computer science*.

[14] R. Li K-T. Fang and A. Sudjianto. 2005. *Design and Modleing for Computer Experiments*. Chapman and Hall/CRC.

[15] Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. 2018. Autotune. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining* (Jul 2018). https://doi.org/10.1145/3219819.3219837

[16] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E Long. 2017. 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)*. New York, NY, USA, Article 42, 14 pages.

[17] T. Miyazaki, I. Sato, and N. Shimizu. 2018. Bayesian Optimization of HPC Systems for Energy Efficiency. In *High Performance Computing*. Springer International Publishing, Cham, 44–62.

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

[19] C. D. Gelatt S. Kirkpatrick and M. P. Vecchi. 1983. *Optimization by Simulated Annealing*. Vol. 220. Science.

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