auto-tuning of computer systems using black-box optimization : an application to the case of i/o accelerators

Sophie Robert

université PARIS-SACLAY

Auto-tuning of computer systems using black-box optimization: an application to the case of I/O accelerators

Auto-optimisation de systèmes informatiques à l'aide de méthodes d'optimisation de boîte noire : une application au cas des accélérateurs E/S

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique
Unité de recherche : Université Paris-Saclay, UVSQ, LIPARAD, 78180, Saint-Quentin en Yvelines, France.
Référent: : Université de Versailles-Saint-Quentin-en-Yvelines.

Thèse présentée et soutenue à Paris Saclay, le 12/11/2021, par

Sophie ROBERT-HAYEK

Composition du jury:

Nicolas VAYATIS Professeur, Ecole normale supérieure Paris-Saclay, FRANCE	Président
Jean-François Méhaut Professeur, Université Grenoble Alpes, FRANCE	Rapporteur
Maria PEREZ HERNANDEZ Professeur, Universidad Politécnica de Madrid (UPM), ESPAGNE	Rapportrice
Christophe DENIS Maître de conférences (HDR), Sorbonne Université - LIP6, FRANCE	Examinateur
Mathilde MOUGEOT Professeur, ENSIIE, ENS-Paris Saclay , FRANCE	Examinatrice

Direction de la thèse:

Soraya Zertal Dr, Li-Parad	Directrice de thèse
Philippe Couvée Atos R&D BDS Data Management	Co-encadrant
Grégory Vaumourin Dr, Atos R&D BDS	Invité

Thèse de doctorat

NNT: 2021UPASG083

Even if I could have all knowledge and understand every mysteries, if I do not have love, I am nothing.
— Paul of Tarsus

Acknowledgements

I would like to thank from the bottom of my heart my thesis advisor, Dr Soraya Zertal, for her guidance, dedication and unfailing support throughout those three years. Your strength, perseverance and optimism have kept me going through hardships and victories. Thank you for teaching me that success is born out of hard work, determination and discipline.

The warmest of thanks goes to Philippe Couvée, whose kindness, patience and wisdom have been a beacon of light on my path. I could never express how thankful I am for the trust, the freedom and the support you have given me for this research. Thank you for teaching me that wisdom is born out of kindness, generosity and understanding.

I could never thank enough Dr Grégory Vaumourin for both his contributions to my work and his unfailing moral support. Without you, both this manuscript and I would probably be very different today, and I am grateful you accepted to join me for this adventure. Thank you for teaching me that the best ideas are born out of observation, wit, and rigor.

A huge thanks goes to Dr Gaël Goret, and the contributions he has made to this thesis. I definitely would not be here if we hadn't crossed path. Thank you for your trust in me. A big thanks as well to my soon-to-be colleagues from the Data Management team, as well as to Virginie Mégy, for all of their help and support throughout these years.

A special thought goes to my grandparents, André and Maylis, who have welcomed me with open arms whenever I needed it, and to Françoise and Jean-Paul as well, who have always trusted me and my abilities. Thank you for being such role models. Thanks to my mom as well, for supporting and encouraging me however she could, and for my little sister, who's always been there whenever I needed her.

I would also like to thank all of my friends, those of old and those I've made along the way. You have all shaped me by your love and your support, and I can't ever express how glad I am to have had so many listening ears throughout these years.

Of course, my last thanks goes to Guillaume, but I cannot find words that would do justice to how much you have helped me, so a simple thank you will have to do :).

I'll keep these last words to keep in mind all those that were not as lucky as I was during the dark times of this pandemic. My heart goes out to all those that are fighting, or have lost the fight, to the Coronavirus. May we all come out of this crisis better people than when we went in, with a new sense of responsibility for others and for the world.

Contents

List of Figures

List of Tables

Introduction

Most of the software of modern computer systems come with many configurable parameters that control the system's behavior and its interaction with the underlying hardware. These parameters are challenging to tune by solely relying on field insight and user expertise, due to huge spaces and complex, non-linear system behavior. Besides, the optimal configuration often depends on the current workload, and parameters must be changed at each environment variations. Because of this, users often have to rely on the default parameters given by the vendor, and do not take advantage of the possible performance of running their application on a tuned system. The more complex these systems are, the more important the tuning becomes, as the components interact which each other in ways that are hard to grasp by the human mind. As architecture becomes more and more service oriented, the number of components per system increases exponentially, along with the number of tunable parameters. This problem is particularly observed within High Performance Computing (HPC) systems, as the hundreds of devices assembled to make supercomputers create very complex and highly configurable stacks. As performance is the major concern in this field, each component must be fine tuned, which is almost impossible to achieve solely through field expertise.

Faced with the inability of relying solely on users to take adequate decisions for the parametrization of complex computer systems, new tuning methods have emerged from various computer science communities to automate parameter selection depending on the current workload. Because they do not require any human intervention, these approaches are commonly called *auto-tuning* methods, a term which encompasses a broad range of methods taken from the optimization and machine learning field. Throughout the years, they have been successfully applied to a wide range of systems, such as storage systems, database management systems and compilers.

This thesis explores a subset of auto-tuning methods called *black-box optimization*, designed to optimize functions without making any assumption on their properties, hence the term of black-box, and discusses their applicability to different tunable HPC systems. Throughout this manuscript, we present different black-box optimization heuristics, make them resilient to the interference caused by the HPC stack's complexity, design a mechanism for automatic parametrization upon application's submission

and make them available to the scientific community as an Open Source software. While the presented methods are designed to be generic, they have been tested and proven to be efficient on three very different components of the HPC stack: a mixed software/hardware appliance called Smart Burst Buffer (SBB), a smart prefetching strategy called Small Read Optimizer (SRO) and the Modular Component Architecture (MCA) parameters of the Message Passing Interface (MPI). This thesis thus provides contributions to several fields, each related to a different aspect of systems' auto-tuning.

The content of this manuscript is organized as follows. Chapter 1 gives the general context of this work, by giving insight on the different methods for system's parametrization and the conceptual choices made for our auto-tuner, as well as the main related works available in the state-of-the-art. We also use this chapter to lay the problem formulation and the mathematical notations used throughout the thesis.

In chapter 2, we present the two main configurable systems used to test the developed auto-tuner. Called Input/Output (I/O) accelerators, these two components aim at reducing the I/O bottlenecks by speeding up HPC systems write and read operations. This chapter presents them and their main functionalities, justify their relevance for auto-tuners testing, and introduces the main tuning benchmarks. These accelerators will be used throughout the thesis in order to prove the relevance of the developed methods and the efficiency of our tuning methodology.

Chapter 3 introduces the main principles of black-box optimization and explains three of the most commonly used heuristics we have selected as the most promising for tuning. We also provide in this chapter the different criteria we will use to quantify the behavior and the relevance of each method.

In chapter 4, we use these criteria to perform a comparison of the performance of three selected heuristics for each of the I/O accelerators in a controlled environment, in order to select the most promising one for our tuner. These experiments allow us to successfully prove the relevance of black-box optimization for tuning HPC components, as well as select the most promising optimization method that will be used in the remainder of the work.

Chapter 5 focuses on the improvement of the resilience of these heuristics to the noise often present on shared systems, such as HPC systems. As the performance measured on the system can represent its occupancy rather than the impact of the tuned systems parameters, heuristics must be able to handle noise and still provide adequate tuning results in a disturbed environment. For this reason, we explore and improve resampling strategies commonly found in the field of stochastic optimization, by suggesting a new resampling algorithm specifically tailored to the tuning of HPC systems. This algorithm is evaluated in different noisy conditions when tuning the two I/O accelerators, and provides a significant improvement over state-of-the-art resampling strategies.

In chapter 6, we present a generic Open-Source optimization framework bundling all the developed methods in order to make them easily available to the rest of the scientific community. This framework provides all the methods used in the optimization experiments of the previous chapters. It also comes with a generic interface for registering tunable components and launching optimization experiments to find their best configuration. A Web Interface is provided as well for results visualization of on-going and finished experiments.
Chapter 7 presents a possible integration of our work for automatic parametrization of incoming applications, by using a matching pipeline on users submission. This pipeline relies on record linkage methods, often used on medical data, but to our knowledge not yet in the computer science field. These methods find the optimization experiment closest in terms of metadata, and use the best parameters found during this experiment. We demonstrate the usefulness of such a mechanism in an environment mimicking conditions seen in production, and prove that our auto-tuner is ready to bring a significant improvement in performance compared to already existing solutions.
Finally, chapter 8 widens the scope of our work by optimizing the parameters of the de-facto standard library for parallel programming Message Passing Interface (MPI), using Bayesian optimization. We show the wide applicability of our work by finding a significant improvement in performance compared to using grid search or the default parametrization for the popular OSU benchmarking suite.
A summary of the different results as well as the major contributions per field are available in the conclusion in chapter 9. We also present in this chapter the on-going works at the time of redacting this thesis, as well as the short term and long term perspectives that could improve upon this work.

As each of these chapters introduce methods from different fields and different scientific communities, we have made the choice to make each chapter as self-contained as possible, and consequently organize the bibliography on a per chapter basis. All of the acronyms used throughout the thesis are available in appendix A. A glossary gathering the definitions spread out throughout the chapters is available in appendix B. The list of peer-reviewed publications and patents related to this thesis is available in appendix C.

Chapter 1

Auto-tuning of computer systems

Contents

As mentioned in the introduction, most modern software and hardware components come with a wide range of parameters that have a high impact on their behavior and the system's performance. Finding the optimal parametrization for each application on a system is key to make the most of its resources and has been for a long time a topic of interest in the computer science community. This problematic is particularly important in the field of High Performance Computing (HPC) where maximizing the systems' performance and resource consumption is of the utmost importance. Indeed, as supercomputers handle a wide range of applications with heavy computation and Input/Output (I/O) workloads, maximizing their performance requires tuning their different components with the most adapted parametrization for the currently running application. In this chapter, we explore different methods for finding the optimal parametrization of computer systems and especially supercomputers, present the approach we have selected for this thesis, and describe some works related to ours with similar goals.

1.1 Taxonomy of parametrization methods

Several approaches can be considered to set the parametrization of a configurable computer system.

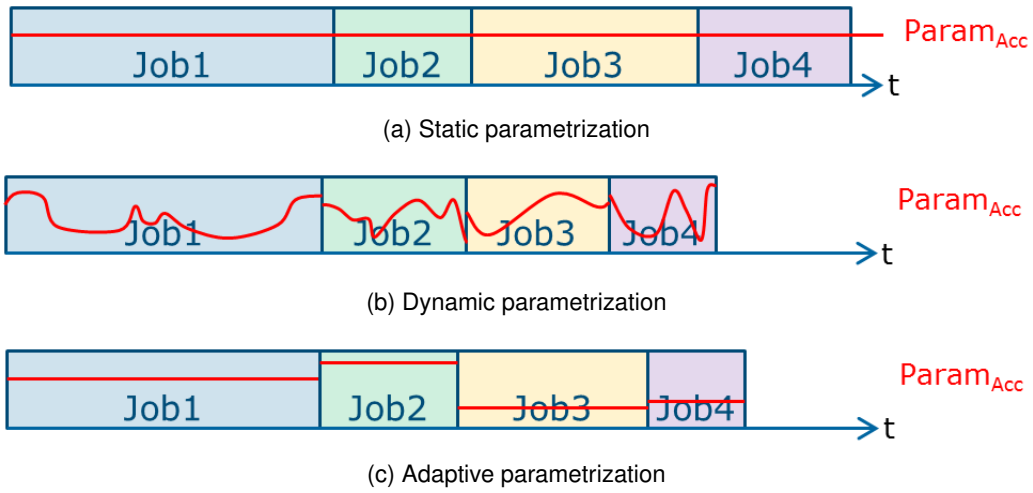
(a) **Static parametrization**: it consists in giving a static (default) parametrization for every application run on the system, as described in figure 1.1a. It is the type of parametrization most often encountered with commercial software. While easy to implement, it can often cause suboptimal performance because no single parametrization can match the requirements of a great diversity of applications and their workloads.

(b) **Dynamic parametrization**: it consists in giving at run-time different parametrization for each timeframe of the workload execution and dynamically adjust the parameters depending on the behavior of the application, as illustrated in figure 1.1b. While this can be considered as the optimal model, it has several drawbacks. It indeed requires a perfect knowledge of the tuned system, knowledge that is often hard (if not impossible) to acquire because of the complexity of the components and their many interactions with other parts of the system. It also requires a very flexible and dynamically parametrizable component able to change its parametrization at run-time. This makes this solution impossible to implement when dealing with many commercial closed-source software that do not allow to change their parametrization at run-time and which code cannot be modified. Even in the case of open-source and modifiable software, it has a high development cost and requires strong development skills. Such a type of parametrization also requires some kind of monitoring system, in order to make real-time decisions on the changes of the parameters. This makes this parametrization more intrusive and the solution more difficult to deploy on large systems. Another drawback of this parametrization is the possible competition between the optimization method, the monitoring system and the application, which can degrade the application's performance.

Examples of tuning frameworks relying on such type of parametrization are the COLT self-tuning framework [42], which dynamically adjusts databases configuration by continuously monitoring incoming queries, the MROnline framework [34] that dynamically selects the best configuration for Hadoop Yarn, depending on the characteristics of the MapReduce application, and the mARGOt [44] auto-tuning plugin which uses a feedback loop to tune the system and the running application, in order to optimize HPC clusters energy consumption.

(c) **Adaptive parametrization**: it is a trade-off between static and dynamic tuning and it consists in selecting a parametrization adapted to each running application, and keeping it con-

stant throughout the run, as described in figure 1.1c. We refer to this parametrization as adaptive parametrization [33]. It avoids several of the drawbacks mentioned for the other types of parametrization. It is indeed more efficient than static parametrization, because it takes into account the specificity of each application, and it does not have the drawbacks of dynamic parametrization: there is no resource concurrency with the optimized application as the computations associated with the optimization process can be performed between application runs and in the vast majority of cases, it does not require any real-time knowledge through a monitoring system. It also does not require changes for the component in terms of additional developments or modifications. Additionally, it has lighter requirements in terms of knowledge and insight on both the application and the component, as the application must be optimized as a whole instead of on a real-time basis. It is also particularly interesting for components which are often run with the same workloads, as is the case for HPC applications that are frequently launched on similar datasets. Because of these advantages, we have selected adaptive parametrization as the parametrization method used for auto-tuning throughout this thesis.



(a) Static parametrization

(b) Dynamic parametrization

(c) Adaptive parametrization

Figure 1.1: Different possible parametrization approaches

1.2 Finding the optimal parametrization of computer systems

Once adaptive parametrization has been settled as the chosen method for the parametrization of the system, an optimization strategy must be adopted to find the best parametrization of a system for any incoming application, given its characteristics. Several strategies can be considered, and we focus on four of the most commonly found in the state-of-the-art.

(a) **Exhaustive sampling**: this strategy is the simplest, and consists in performing an exhaustive

sampling (also called grid search) of the parametric space by testing for each parametrization its impact on the performance function of the application. After testing every single parametrization, the parametrization associated with the optimal performance can be selected as the best parametrization which should be used for the application. However, the large size of the parametric space and the long duration of most applications, especially in the HPC field, makes this option very time consuming and thus impractical in most cases, as it would indeed require several days or even weeks of non-stop computations: for example, for a benchmark lasting only one minute, it would take about a week to evaluate a hundred parameters in four dimensions, which corresponds to a rather small parametric space in the computer science field.
Exhaustive sampling is often the only tuning methods shipped with softwares, as it is the case for the two major Message Passing Interface (MPI) implementations Intel MPI [2] and Open MPI [4], respectively with the mpitune [3] and opto [15] tools.

(b) **Analytical modeling**: another alternative is the design of a theoretical explicit model for the software or hardware to be tuned which would describe the relationship between the parameters, the incoming workload and the performance function. Such analytical model would enable to select the best parameters given the profile of the submitted workload by predicting the behavior of the component depending on the workload's characteristics. However, building such a model is often a difficult task because of the required precise insight on the behavior of the tuned component and its impact on the running applications. It is often hard to gather such precise knowledge on the impact of the component and to model the effect of each of the parameters on the behavior of the running application, because we are limited in our understanding of the actions of the component, especially when dealing with closed-source software. Another difficulty is the complexity of the interactions between each sub-part of a computer system when running in production. To tackle this obstacle, analytical modeling has to make some strong assumptions on the system and has to simplify or overlook some of the interactions of the system. Finally, the strongest argument against theoretical modeling in the context of this thesis is the specificity of such model to a given component, and the impossibility to re-use this model for tuning a different kind of component. As we are working with very different tunable components which behavior differs depending on the topology and the application, as described in chapter 2, building an analytical model for each case would be impractical and would hinder us from proposing an universal tuning framework.

Many different analytical models developed to predict systems' performance are available in the literature. For example, Eller et al. have suggested in [19] models to describe the behaviors of

HPC applications running at scale, and use these models to speed up scientific solvers. Hoefler et al. describe in [24] major requirements and steps for simple performance modeling, in order to characterize the behavior of applications. In [46], Tung et al. build an analytical model to select optimal buffer parameters for database systems and use this equation to improve the performance of a PostgreSQL database. Communication models to estimate optimum parameters are also popular in the MPI community such as the Hockney [23], LogP [16], LogGP [5] and PLogP [31] models. To find the optimal parametrization of burst buffers Input/Output (I/O) accelerators, one of the major goal of this thesis, Aupy et al. have suggested to use different analytical models. In [6], they use a Markov chain based model to describe the behavior of the burst buffer, and in [6], a polynomial time algorithm to predict optimum cache size in the case of static and dynamic resource allocation. In [40], Schenk et al. model the behavior of the IME [1] burst buffer and use this model to infer its required specifications and capacities to prevent I/O contention.

(c) **Machine Learning models**: it consists in using Machine Learning models to learn the relationship between the system's performance and the system's parameters, either by using regression or classification models. While this approach has shown good results for several use-cases, the models are strongly dependent on the dataset used to train it as well as on the underlying hardware. Generating the dataset may introduce bias in the resulting model, and changing the hardware may compromise the correlation between parameters and the relative effects found by the model. It also often requires to use a profiling system, in order to gather some data characterizing the workload to build some features to train the model on. Additionally, they share the same drawback of analytical models as they are also component dependent, and a new Machine Learning model has to be trained for each new tunable system. However, one of their main advantage is their ability to predict values for unseen instances. Machine Learning based approaches have been used for the tuning of the parameters of a wide range of computer systems.

Examples of such models can be found in the MPI community, as Hunold et al. build machine learning models to predict the optimum algorithm configuration in their works [25] and [26]. Several Machine Learning optimization models have also been suggested in the I/O community, for example through the works of Madireddy et al. in [37], who model I/O performance of an application using the file system's and the application's characteristics using Gaussian Processes. In [8], Bagbaba trains random forests to predict the optimal configuration of the ROMIO MPI-I/O implementation. A similar task is undertaken in [28] by Isaila et al., who use a combination of surrogate and analytical models to optimize ROMIO. In [45], Sun et al. use random forests to predict the performance of an application given its profile and MPI parameters. Other examples

can be found in the DataBase Management Systems community, where the OtterTune [48] and Baloo [21] frameworks perform auto-tuning by fitting regression models to predict the database performance as a function of its parameters. The OtterTune framework has been successfully applied for the tuning of MySQL, PostreSQL, Actian Vector, as well as an instance of FoundationDB [41].

(d) **Black-box optimization**: it consists on relying on black-box optimization methods to find the system's optimal parametrization. When used for auto-tuning of computer systems, black-box optimization consists in treating the combination of the configurable component and the running application as a black-box, which takes as input the parametrization of the tuned component and outputs a performance measure, as depicted in figure 1.2.

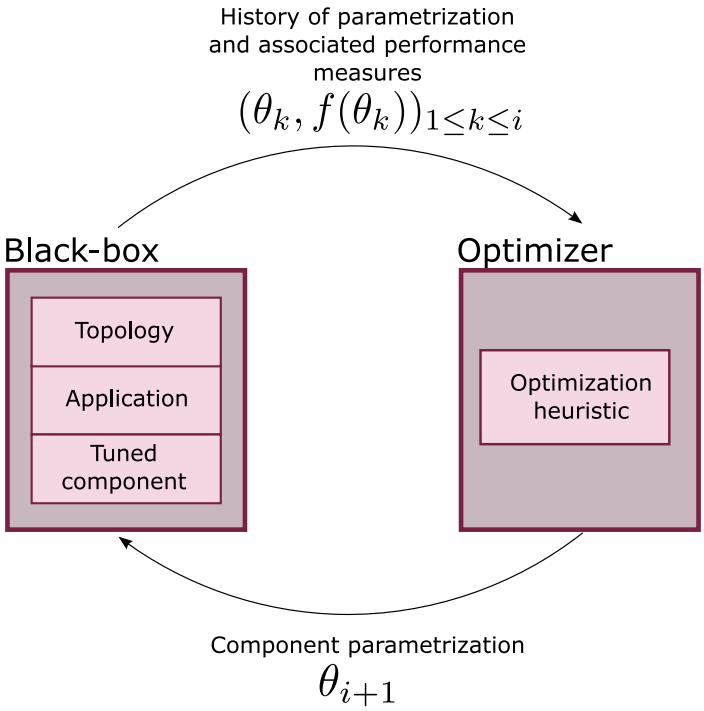


Figure 1.2: Schematic representation of the black-box

Given this new information, the optimization heuristic decides the next parametrization to evaluate by the system. This feedback loop is repeated until a convergence criterion is reached and the best found parametrization is deemed to be the best parametrization for the system. By design, black-box optimization is very adapted to our use-case as it avoids several of the drawbacks mentioned above. It is indeed designed to work in a minimal number of iterations, thus limiting the computation time compared to exhaustive sampling, and makes no assumptions on the optimized function or on the parametric space, in opposition to building an analytical model. It also has no dependence on a previously generated dataset. However, this great adaptability

comes at the cost of running the same application several times, which has a non-negligible resource cost. Unlike machine learning and analytical models who give instant prediction once built, the black-box optimization methods are not suited on their own to transfer knowledge between applications, even though some mechanisms can be built to still benefit from previous experiences. However, the wide diversity of applications and tunable components in the HPC domain makes us favor adaptability and optimization quality rather than prediction speed and black-box optimization is the method developed throughout this thesis.

In the literature, black-box optimization is a popular choice for tuning different tunable systems, and it has been particularly helpful in computer science to look for the optimal configurations of various software and hardware components. In the field of Big Data Processing systems, software that are notoriously hard to tune such as Hadoop, Spark and Storm, have benefited from black-box optimization. It is for example the methodology chosen by Liao et al. in their Gunther framework [36] which relies on Genetic Algorithms to find the optimal parametrization of Hadoop. In [29], Jamshidi et al. use Bayesian Optimization to optimize the stream processing system Storm. In [17], Desani et al. 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.

Database Management Systems (DBMS) tuning relying on surrogate modeling with expected improvement and pruning strategies have been suggested through the iTuned framework designed by Duan et al. in [18]. This framework has significantly improved the performance of a PostgreSQL for different workloads. In [49], Xi et al. have used a variation of Simulated Annealing with local estimation to find the optimum configuration of Web servers. Using black-box optimization to optimize query has also proven its efficiency from the start in [27] using simulated annealing and [11] using genetic algorithms.

Storage systems are also hard to tune, as they have both a very large parametric space, with a strong dependence on the running workload [13] [35]. The efficiency of black-box optimization for tuning storage systems when faced with different workloads has been explored by Cao et al. in [14], where they propose 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 by Li et al. in [35] and an optimal parametrization for the several layers of HDF5 library was found using genetic algorithms by Behzad et al. in [10]. An extension of this auto-tuner which selects the best parameters according to the I/O pattern is described in [9] by the same authors.

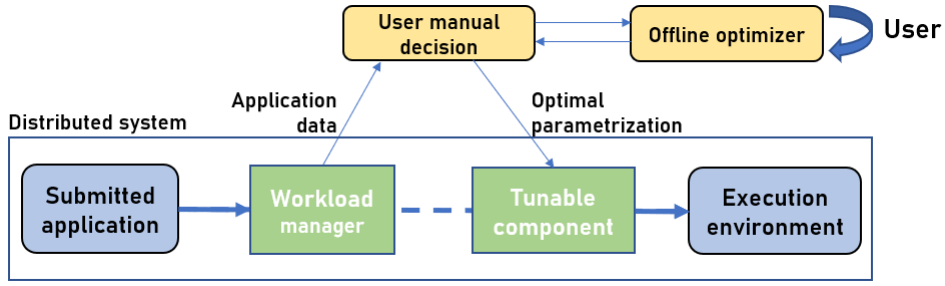
Within the HPC community, auto-tuning has gained a lot of attention for tuning particular HPC application and improve their portability across architectures [7]. In [43] and [32], Seymour et al. and Knijnenburg et al. provide a comparison of several random-based heuristic searches (Simulated annealing, genetic algorithms ...) that have provided some good results when used for code auto-tuning. Bergstra [12] has had good results as well in this field with surrogate modeling using boosted regression trees. In [38], Menon et al. use Bayesian Optimization and suggest the framework HiPerBOT to tune application parameters as well as compiler runtime settings. HPC systems energy consumption can also benefit from Bayesian Optimization, as Miyazaki et al. have shown in [39] that an auto-tuner based on a combination of Gaussian Process regression and the Expected Improvement acquisition function has raised their cluster to the Green500 list. A scheduling algorithm using genetic algorithms has been introduced by Kassab et al. in [30] and manages to schedule jobs under a limited energy power constraint. The MPI community has also shown the superiority of a hill-climbing black-box algorithm over an exhaustive sampling of the parametric space in [20] and [47].

1.3 Online and offline tuning and their variations

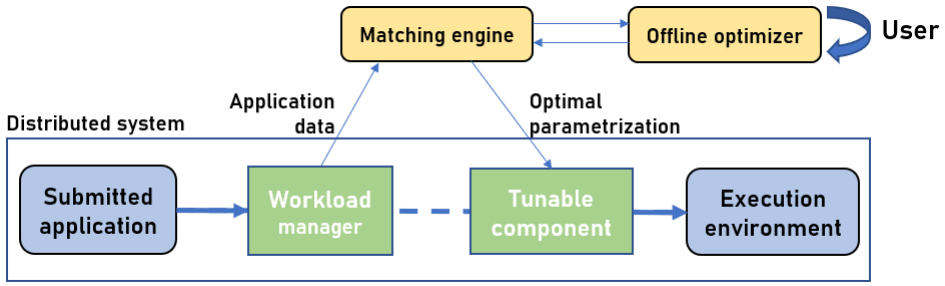
As mentioned in the previous section, one of the major drawbacks of black-box optimization is its inability to transfer knowledge between optimizations when used on its own. However, there exists several methods to address this issue and share information between runs, such as adding an online tuning component to the application's submission process or making the optimization process incremental rather than all at once. This type of tuning is called online because of its dynamic interactions with the cluster running in production, in opposition to offline tuning which is isolated from production runs. Several distinctions can be made on the level of interactions between the auto-tuner and the cluster's applications' submission mechanism.

(a) **Offline tuning**: Finding the optimal parametrization of the component can take place *offline*, which consists in finding the optimal parametrization for the workload without the system running in production, as described in figure 1.3a. This can be considered as a setup phase as described by Behzad et al. in [10], which consists in finding the optimal parametrization for a given component, topology and hardware when installing the machine or using a new application. Whenever this application is launched again the user must manually set the parametrization, a process which can be cumbersome and prone to human mistakes. Most tuners described and available in the literature are designed for offline tuning [14] [25] [26].

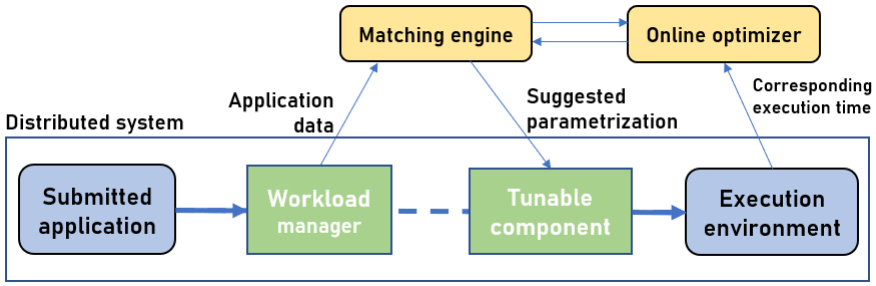
(b) **Online tuning**: Another solution is to perform the tuning *online*, by performing the tuning at



(a) Offline tuning



(b) Trade-off between offline and online tuning



(c) Online tuning

Figure 1.3: Different possible variations of online and offline tuning

each submission run and accumulating knowledge along with the applications executions. At each submission, the application is analyzed through a recognition engine. If the application is recognized as having already been run on the system, it is assigned the parametrization predicted by the optimization heuristic given its previous history. If not, a new entry is initialized for this application and the application is run using a parametrization suggested through the initialization plan. The execution time corresponding to the selected parametrization for this application is then recorded in the tuning database, as described in figure 1.3c. This type of method is described in [33], but not implemented in practice. Its main drawback is the strong dependence of the success of the tuner on the performance of the recognition engine, as well as the additional strain put on evaluation metrics, as is presented in table 3.1 of chapter 3.

(c) **Trade-off between offline and online tuning**: A trade-off between fully offline or fully on-

line tuning can be reached, by running the optimization process offline and gathering knowledge from the input data points and the associated measured performance function, as well as the characteristics of the application in terms of metadata. This type of tuning is illustrated by figure 1.3b. It builds a side database with the tuning information for each application that has been optimized offline, consisting of the best parametrization for an application, as well as information regarding of its metadata or traces. Every user application submitted to the system goes through a recognition engine similar to the one described in the online tuning section, 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 parametrization is selected either using initialization techniques or the default. While the optimization process happens offline, the exploitation of the results happens online because of this recognition of the incoming application.

Another possible method, relying on the identification of already seen workflow, is suggested in [13], but to our knowledge has not yet been implemented and tested in real-life conditions. The OtterTune framework [48] also provides a similar mechanism by linking unknown workloads to those closest in terms of their euclidean distances, but it requires *in situ* metrics describing the activity of the application in real-time and cannot be achieved using only *a priori* submission data. A similar approach is suggested in [22], where Gur et al. train a multi-model reinforcement learning on different database workloads, and select which model to use depending on the cosine distance between the current workload and already known ones.

Throughout this thesis, we remain aware that auto-tuners can be used either offline and online, or a mix of both. This distinction affects the evaluation metrics, and are detailed in section 3.5. Methods using a trade-off between offline and online tuning are explored in chapter 7, where a recognition engine based on metadata is implemented and evaluated.

1.4 Problem formulation

To finish this chapter, we introduce the various notations that will be used throughout this manuscript. As the problem of system tuning is very general, we propose notations as generalist as possible in order to not make our methodology problem dependent. The problem can be transcribed formally as follows:

Let S the system to optimize and θ_i its parametrization, which belongs to a discrete subset of the possible parametrizations (which might be infinite) $\mathbb{\Theta} = \{\theta_i\}_{i \in \mathbb{N}}$. Let $\mathcal{A}$ the job for which we choose to optimize the system. Let f_s be the performance function associated with the application and the

system.

$$f_S : (\mathcal{A},\ \Theta) \longrightarrow \mathbb{R} \tag{1.1}$$

The optimization problem that we are trying to solve can be summed up as finding:

$$\text{argmin}_{\theta_i \in \Theta} f_{S,\mathcal{A}}(\theta_i)$$

For our use-case of working with HPC systems, the system S to optimize is represented by the combination between the component to tune and the execution context, which corresponds for example in the HPC context to the compute nodes and the storage system characteristics. This execution context is considered to be set throughout each run of the experiment in order to make the system S immutable. 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.

1.5 Conclusion

In conclusion, we have introduced in this chapter the problematic of auto-tuning of computer systems, which is the main focus of this thesis. We have presented some key concepts and definitions, as well as major works that are related to ours. The study of the nature of HPC systems points to the selection of a trade-off between dynamic and static parametrization, which we called adaptive parametrization. The careful survey of existing works has led us to select black-box optimization to perform system's tuning, instead of analytical models or Machine Learning models. Finally, the consideration of the difference between online and offline tuning has been discussed and highlighted. These differences in terms of use of the tuner, and the consequences they have on the tuned system, are discussed throughout this thesis, and especially in chapter 7. This chapter has also been the opportunity to introduce the notations and formulations used throughout the manuscript.

In the following chapter, we introduce the two main tuned components used to evaluate the relevance of our selected tuning methodology. They provide us with a practical example to evaluate the performance of the auto-tuner developed throughout this work, because of their diversity, sensitivity to their parameters, and unsuitability of the default parametrization.

—

Bibliography

[1] *Ime: Infinite memory engine*. https://www.ddn.com/products/ime-flash-native-data-cache/.

[2] *Intel mpi: Deliver flexible, efficient, and scalable cluster messaging*. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html#gs.685gg1.

[3] *mpitune: tunes the intel mpi library parameters for the given mpi application*. https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/command-reference/mpitune.html.

[4] *Open mpi: Open source high performance computing*. https://www.open-mpi.org/.

[5] A. D. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, *LogGP: incorporating long messages into the LogP modelone step closer towards a realistic model for parallel computation*, in Proccedings of the Seventh Symposium on Parallelism in Algorithms and Architectures (SPAA '95), 1995.

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

[7] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, *Autotuning in high-performance computing applications*, in Proceedings of the IEEE, no. 11, 2018, pp. 2068 – 2083.

[8] A. Bababa, *Improving collective i/o performance with machine learning supported auto-tuning*, in 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2020, pp. 814 – 821.

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

[10] 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, 2013, pp. 1 – 12.

[11] K. Bennett, M. Ferris, and Y. Ioannidis, *A genetic algorithm for database query optimization*, in Proceedings of the fourth International Conference on Genetic Algorithms, 1991, pp. 400 – 407.

[12] J. BERGSTRA, N. PINTO, AND D. COX, *Machine learning for predictive auto-tuning with boosted regression trees*, in 2012 Innovative Parallel Computing (InPar), 2012, pp. 1–9.

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

[14] 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, 2018, pp. 893 – 907.

[15] M. CHAARAWI, J. M. SQUYRES, E. GABRIEL, AND S. FEKI, *A tool for optimizing runtime parameters of open mpi*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2008, pp. 210 – 217.

[16] D. CULLER, R. KARP, D. PATTERSON, A. SAHAY, K. E. SCHAUSER, E. SANTOS, R. SUBRAMONIAN, AND T. VON EICKEN, *LogP: Towards a realistic model of parallel computation*, in SIGPLAN Not., no. 7, Association for Computing Machinery, 1993, pp. 1 – 12.

[17] D. DESANI, V. G. 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), 2016, pp. 43 – 50.

[18] S. DUAN, V. THUMMALA, AND S. BABU, *Tuning database configuration parameters with ituned*, in Proceedings of the VLDB Endowment, 2009, pp. 1246 – 1257.

[19] P. ELLER, T. HOEFLER, AND W. GROPP, *Using performance models to understand scalable Krylov solver performance at scale for structured grid problems*, in Proceedings of the ACM International Conference on Supercomputing, 2019, pp. 138 – 149.

[20] A. FARAJ AND X. YUAN, *Automatic generation and tuning of mpi collective communication routines*, in Proceedings of the 19th Annual International Conference on Supercomputing, 2005, pp. 393 – 402.

[21] J. GROHMANN, D. SEYBOLD, S. EISMANN, M. LEZNIK, S. KOUNEV, AND J. DOMASCHKA, *Baloo: Measuring and modeling the performance configurations of distributed dbms*, in 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2020, pp. 1 – 8.

[22] Y. GUR, D. YANG, F. STALSCHUS, AND B. REINWALD, *Adaptive multi-model reinforcement learning for online database tuning*, in International Conference on Extending Database Technology, 2021.

[23] R. W. HOCKNEY, *The communication challenge for mpp: Intel paragon and meiko cs-2*, in Parallel Computing, no. 3, 1994, pp. 389 – 398.

[24] T. HOEFLER, W. GROPP, W. KRAMER, AND M. SNIR, *Performance modeling for systematic performance tuning*, in State of the Practice Reports, SC '11, 2011, pp. 1 – 12.

[25] S. HUNOLD, A. BHATELE, G. BOSILCA, AND P. KNEES, *Predicting mpi collective communication performance using machine learning*, in 2020 IEEE International Conference on Cluster Computing (CLUSTER), 2020, pp. 259 – 269.

[26] S. HUNOLD AND A. CARPEN-AMARIE, *Algorithm selection of mpi collectives using machine learning techniques*, in 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 2018, pp. 45 – 50.

[27] Y. IOANNIDIS AND E. WONG, *Query optimization by simulated annealing*, in SIGMOD Record, 1987, pp. 9 – 22.

[28] F. ISAILA, P. BALAPRAKASH, S. M. WILD, D. KIMPE, R. LATHAM, R. ROSS, AND P. HOVLAND, *Collective i/o tuning using analytical and machine learning models*, in 2015 IEEE International Conference on Cluster Computing, 2015, pp. 128 – 137.

[29] P. JAMSHIDI AND G. CASALE, *An uncertainty-aware approach to optimal configuration of stream processing systems*, in arXiv, 2016.

[30] A. KASSAB, J.-M. NICOD, L. PHILIPPE, AND V. REHN-SONIGO, *Assessing the use of genetic algorithms to schedule independent tasks under power constraints*, in 2018 International Conference on High Performance Computing Simulation (HPCS), 2018, pp. 252–259.

[31] T. KIELMANN, H. E. BAL, AND K. VERSTOEP, *Fast measurement of logp parameters for message passing platforms*, in Parallel and Distributed Processing, 2000, pp. 1176 – 1183.

[32] P. KNIJNENBURG, T. KISUKI, AND M. O'BOYLE, *Combined selection of tile sizes and unroll factors using iterative compilation*, in The Journal of Supercomputing, 2003, pp. 43 – 67.

[33] M. N. L. VINCENT AND G. GORET, *Self-optimization strategy for io accelerator parameterization*, in International Conference on High Performance Computing, 2018, pp. 157 – 170.

[34] M. LI, L. ZENG, S. MENG, J. TAN, L. ZHANG, A. R. BUTT, AND N. FULLER, *Mronline: Mapreduce online performance tuning*, in Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '14, 2014, pp. 165 – 176.

[35] 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, 2017, pp. 42:1–42:14.

[36] G. Liao, K. Datta, and T. Willke, *Gunther: Search-based auto-tuning of mapreduce*, in Euro-Par, vol. 8097, 2013, pp. 406 – 419.

[37] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. Wild, *Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems*, 2018, pp. 184 – 204.

[38] H. Menon, A. Bhatele, and T. Gamblin, *Auto-tuning parameter choices in hpc applications using bayesian optimization*, in 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2020, pp. 831 – 840.

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

[40] W. Schenck, S. El Sayed, M. Foszczynski, W. Homberg, and D. Pleiter, *Evaluation and performance modeling of a burst buffer solution*, in ACM SIGOPS Operating Systems Review, 2017, pp. 12 – 26.

[41] T. Schmied, D. Didona, A. Döring, T. Parnell, and N. Ioannou, *Towards a general framework for ml-based self-tuning databases*, in CoRR, 2020.

[42] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, *Colt: continuous on-line tuning*, in Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 2006.

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

[44] C. Silvano, G. Agosta, A. Bartolini, A. R. Beccari, L. Benini, J. Bispo, R. Cmar, J. M. P. Cardoso, C. Cavazzoni, J. Martinovi, G. Palermo, M. Palkovi, P. Pinto, E. Rohou, N. Sanna, and K. Slaninová, *Autotuning and adaptivity approach for energy efficient exascale hpc systems: The antarex approach*, in 2016 Design, Automation Test in Europe Conference Exhibition (DATE), 2016, pp. 708 – 713.

[45] J. Sun, G. Sun, S. Zhan, J. Zhang, and Y. Chen, *Automated performance modeling of hpc applications using machine learning*, in IEEE Transactions on Computers, vol. 69, 2020, pp. 749 – 763.

[46] D. TRAN, P. HUYNH, Y. TAY, AND A. TUNG, *A new approach to dynamic self-tuning of database buffers*, in ACM Transactions on Storage, 2008.

[47] S. VADHIYAR, G. FAGG, AND J. DONGARRA, *Automatically tuned collective communications*, in SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, 2000.

[48] D. VAN AKEN, A. PAVLO, G. J. GORDON, AND B. ZHANG, *Automatic database management system tuning through large-scale machine learning*, in Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 1009 – 1024.

[49] B. XI, Z. LIU, M. RAGHAVACHARI, C. XIA, AND L. ZHANG, *A smart hill–climbing algorithm for application server configuration*, in Thirteenth International World Wide Web Conference Proceedings (WWW'2004), 2004, pp. 10 – 15.

Chapter 2

Tuned systems: I/O accelerators

Contents

While the main context of this thesis described in chapter 1 is generic and auto-tuning can be applied to a wide range of HPC components, we focus throughout this work on the auto-tuning of two systems in particular, called I/O accelerators. In this chapter, we introduce these two I/O accelerators by detailing their purpose and how they operate, then explain why they make good candidates to test an auto-tuning framework. We finally provide some examples of optimization landscapes and the impact of parameters on the performance function.

2.1 I/O accelerators

I/O accelerators are software or hardware components which aim is to reduce the increasing performance gap between compute nodes and storage nodes, that can slow down I/O intensive appli-

cations [6]. On large supercomputers, many nodes performing reads or writes stress the back-end storage and can make the application wait while it performs its I/O, generating I/O bottlenecks. This is especially true for HPC applications that periodically save their current state by performing checkpoints, which causes many writes during a short timeframe [8]. The network between the compute and the storage nodes can become saturated, which slows down the application.

To mitigate these problems, several I/O accelerators have been developed over the years. In this chapter, we present two I/O accelerators developed by Atos: the Small Read Optimizer (SRO) [5] and the Smart Burst Buffer (SBB) [1]. These two accelerators are the main use-cases used to validate the auto-tuning methodology developed in the next chapters. We describe in this chapter their main principles and their tunable parameters, and we identify several accelerable benchmarks and I/O pattern. We quantify and analyze the impact of the accelerators on these benchmarks, in order to justify the strong need of auto-tuning for resource maximization.

2.2 The Small Read Optimizer

2.2.1 General principle

Accessing data *via* a cache can considerably reduce the latency between the computing node and the main storage system. As the data cannot be entirely loaded in cache memory because of its small size, efficient cache management and intelligent scheduling of data movement are necessary.

The Small Read Optimizer (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 them in memory. The file is divided in several zones of size *binsize*, and for each of its zones, the number of accesses is recorded. Once a fixed number of access *cluster_threshold* has been recorded for a given bin, a zone *prefetch_size* is loaded in the cache. The *sequence_length* represents the size of the window of file accesses that is recorded and used by the SRO as it cannot keep track of all accesses made to the file. It works as a FIFO, when a new access is recorded, the oldest access record is deleted from the sequence buffer. Linux already includes a *read-ahead* prefetch strategy that targets sequential reads. SRO targets different I/O patterns as it relies on the access density per file zone, it considers both temporal and spatial aspects. Both prefetcher, SRO and the Linux prefetcher are mostly orthogonals.

Definition 1: Small Read Optimizer

The Small Read Optimizer (SRO) is a dynamic data preload strategy to speed-up pseudo random accesses.

2.2.2 Description of tunable parameters

As detailed in the previous section, the four parameters used by the SRO are:

- **Binsize**: The size 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**: The size of the area that will be prefetched
- **Sequence length**: The number of the previous I/O records kept by SRO

The default values and the type of each parameter are summarized in table 2.1.

Table 2.1: Summary of the Small Read Optimizer tunable parameters

Name	Description	Space	Default	Unit
Binsize	Granularity of the file zone	$\mathbb{N}^*$	10485756	Byte
Cluster threshold	Number of operations to trigger prefetch	$\mathbb{N}^*$	2	N/A
Prefetch size	Size of the prefetched zone	$\mathbb{N}^*$	209715120	Byte
Sequence length	Number of stored I/O records	$\mathbb{N}^*$	100	N/A

2.2.3 Accelerable I/O patterns

As the SRO is designed for applications performing small random read operations, we target applications that perform this kind of operations. We have selected after a careful review 4 I/O patterns that are all sensitive to the accelerator and can be found in industrial applications. In the following paragraph, we describe the process of defining these patterns and their associated benchmarking application.

Pattern generation To generate custom I/O patterns, we use an in-house benchmarking tool called the *fakeapp*. It enables to run applications with very specific I/O patterns to have a fine control over the I/O performed by the application. It 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* and their number *number of leads*. Each lead simulates a process writing the file. Around each lead, random accesses are performed within a certain distance of the lead, called *scatter*. The operations are translated by a window of size *lead advance*. A schematic explanation of leads and scatter is provided in figure 2.1 and the different possible parametrizations of the *fakeapp* benchmark are reported in table 2.2.

Definition 2: Fakeapp

The fakeapp is a benchmarking application developed by Atos to generate complex and custom I/O patterns.

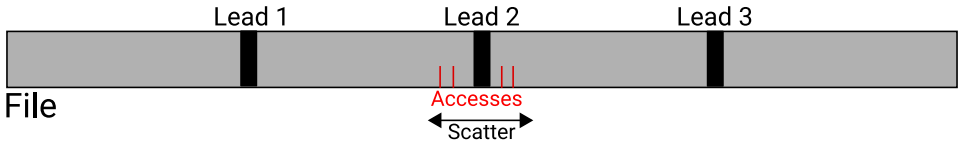


Figure 2.1: Schematic representation of a file for fakeapp benchmarking application

Table 2.2: Description of fakeapp parameters

Parameter name	Parameter flag	Description
Number of operations	N	Number of operations performed by the benchmark
Minimum size of operation	w	Minimum size of performed operations
Maximum size of operation	W	Maximum size of performed operations
Scatter size	S	Scatter width around each lead
Lead	L	Number of conceptual chunks the file will be divided into
Lead advance	l	The sliding window between operations relative to the lead
Repeat	r	Number of repetitions of operations per lead
Lead scatter interval	g	When set, overrides the scatter parameters to assign a random value to each lead's scatter
Lead repeat interval	G	When set, overrides the number of operations parameter so that each lead performs a random set of operations

Studied patterns After analysis of the prefetch mechanism of the accelerator and empirical tests, we have selected four types of I/O patterns that are sensitive to the accelerators. These patterns only give a general idea of the I/O distribution within the file, and can have a wide variation in terms of the values of parameters other than the I/O distribution, such as the number of operations, the size of the operations, the size of the file ... These patterns can be described as follows:

(a) **Fixed number of leads, a sliding window, no scatter and no repeat**: the number of leads L is fixed and operations are performed according to a non-zero lead advance l and the scatter is set to zero. This type of benchmark can be for example representative of an application performing the reading of a matrix stored per row and accessing it by a single thread using its columns.

(b) **Fixed number of leads, a sliding window, a non-zero scatter and a repeat**: the number of leads L takes a small value and after each operation, the following ones are translated with a non-zero lead advance l. The scatter is set to positive, so the operations are taken randomly around each lead location. This type of benchmark can be for example representative of an application performing the reading of a matrix stored per row as well, but accessed by several OpenMP threads by its columns.

(c) **Fixed number of leads, no sliding window, scatter and no repeat**: the number of lead L takes a large value and there is no lead advance after each operation. This type of benchmark is representative of the behavior of a 3D modeling application, and exploring more in depth the interesting zones of this model.

(d) **Fixed number of leads, no sliding window, random scatter per lead and random number of operations per lead**: the number of lead L is fixed and the size of the scatter S is drawn according to a random uniform law, as well as the number of operations per lead. This type of benchmark is a generalization of the previous one, for example representative of a 3D modeling application but with a varying size of the explored zones around points of interest.

In the following chapters, whenever performing tuning of the SRO accelerator, we will be tuning variations of these I/O patterns.

Impact of the accelerator on these patterns A variation of the patterns described in the previous paragraphs are run with a wide set of different parameters and the impact of these parameters on the execution time are reported in figure 2.2 for pattern 1, figure 2.3 for pattern 2, figure 2.4 for pattern 3 and figure 2.5 for pattern 4. The 4 graphs of each figure form a projection of the 4D parametric space. More details on this experiment and the run conditions are given in chapter 4.

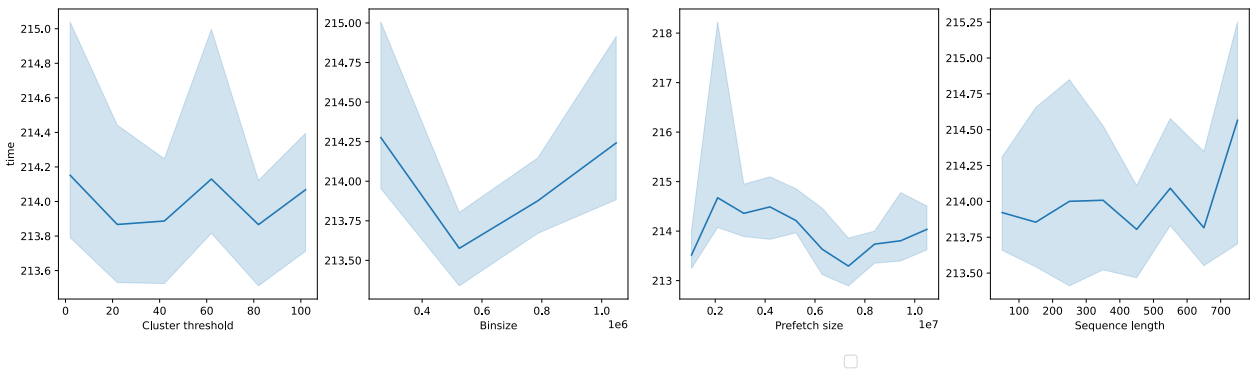


Figure 2.2: Impact of the Small Read Optimizer parameters on pattern 1 benchmark

From these experiments, we can see that the different parameters have a strong impact on the performance of the SRO, and the impact of these parameters is dependent on each I/O pattern. For example, we can see that for pattern 1, represented in figure 2.2, parameters have little influence on the execution time, while in the case of pattern 2, the parameters have a strong influence, as showed in figure 2.3. The individual impact of each parameter is also pattern dependent, as we can see that between pattern 3 (in figure 2.4) and 4 (in figure 2.5), the cluster threshold has an opposite impact on the performance. These examples also highlight the importance of using the best parameters, as the

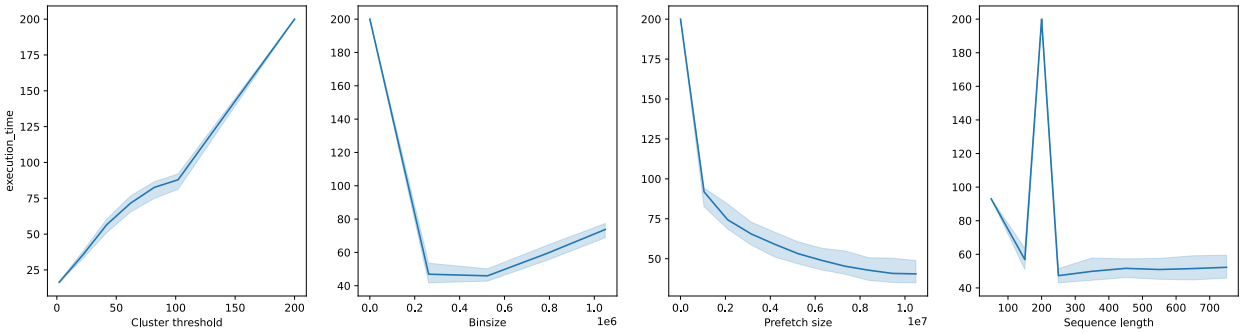


Figure 2.3: Impact of the Small Read Optimizer parameters on pattern 2 benchmark

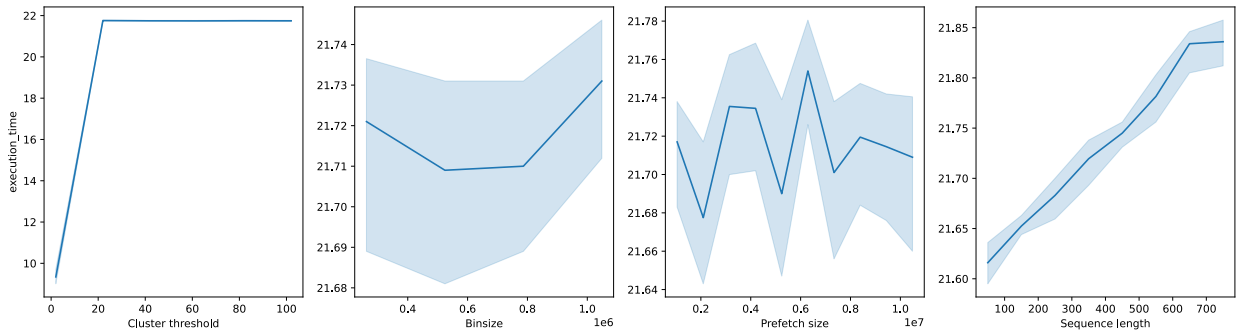


Figure 2.4: Impact of the Small Read Optimizer parameters on pattern 3 benchmark

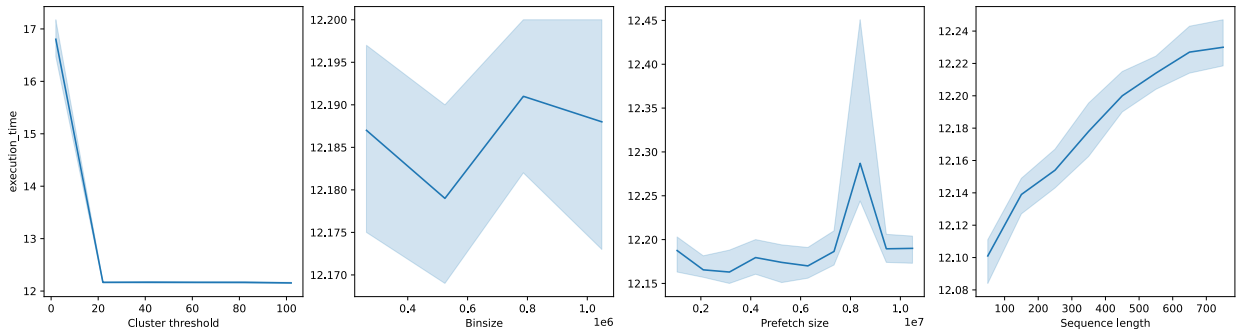


Figure 2.5: Impact of the Small Read Optimizer parameters on pattern 4 benchmark

difference between the default parameters and the optimum parameters range from 21% for pattern 1, to 33% for pattern 4, to 94% for pattern 3 and 95% for pattern 2.

2.3 The Smart Burst Buffer

2.3.1 General principle

The Smart Burst Buffer (SBB) is Atos' implementation of a burst buffer. Burst-buffers have emerged as a promising solution to cope with limited I/O bandwidth and improve the system's performance [7] [10] [11]. Several commercial implementations of burst buffers have been developed by commercial vendors, such as DDN's Infinite Memory Engine [3], Cray's DataWarp [9] [4] and Atos' Smart Burst Buffer [1]. Their purpose is to allocate 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 burst buffer absorbs burst of I/O and asynchronously performs the I/O on the permanent storage bay to reduce the job's stalling time and accelerate its execution.

Definition 3: Smart Burst Buffer

The Smart Burst Buffer (SBB) is Atos' implementation of a burst buffer, an I/O accelerator which consists of a large and fast intermediate node equipped with storage resource, positioned between the compute nodes and the permanent storage system, to increase I/O throughput.

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. A simplified view of an I/O going through the SBB is described in figure 2.6. We can summarize its workflow into 4 steps:

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 threads. The data file is kept in the NVME cache in case of data reuse.

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

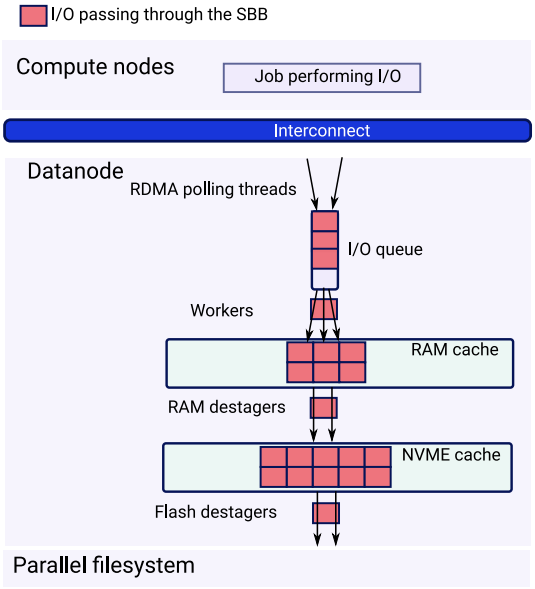


Figure 2.6: Workflow of I/Os passing through SBB

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 room in the caches to receive the next I/O and do not block the application.

Definition 4: Data node

A physical node which part of its cores, memory, persistent storage are dynamically allocated to act as an I/O proxy.

2.3.2 Description of tunable parameters

The eight parameters used by the SBB are:

- **RDMA polling threads**: the number of threads used to handle RDMA transactions between the compute and datanode.
- **Number of workers**: the size of the worker threads pool used to insert incoming I/O into the RAM cache.
- **Number of RAM destagers**: the size of the destager threads pool used to transfer data from the RAM cache to the NVME cache.
- **Number of flash destagers**: the size of the destager threads pool used to transfer the data from the NVME cache to the parallel filesystem.
- **RAM cache size**: the size of the allocated RAM cache on the data node.

- **NVME cache size**: the size of the allocated NVME cache on the data node.
- **RAM threshold**: the fraction of used RAM cache before the process of removing clean data from the RAM cache to free space for the incoming I/O is started.
- **NVME threshold**: the fraction of used NVME cache before the process of removing clean data from the NVME cache to free space for the incoming I/O is started.

The default values and the type of each parameter are summarized in table 2.3.

Table 2.3: SBB tunable parameters

Name	Description	Space	Default	Unit
Number of workers	Number of threads used to transfer data from compute nodes to data node	$\mathbb{N}^*$	10	N/A
RDMA polling threads	Number of threads used to handle RDMA transactions	$\mathbb{N}^*$	2	N/A
Number of RAM destagers	Number of destagers threads used to transform the data from the RAM to the NVME	$\mathbb{N}^*$	10	N/A
Number of flash destagers	Number of destagers threads used to transform the data from the NVME to the parallel filesystem	$\mathbb{N}^*$	10	N/A
RAM cache size	Size of allocated RAM on the data node	$\mathbb{N}^*$	N/A	Gigabyte
NVME cache size	Size of allocated NVME on the data node	$\mathbb{N}^*$	N/A	Gigabyte
RAM threshold	Fraction of used RAM cache before removing clean data	$\mathbb{N}^*$	80	%
NVME threshold	Fraction of used NVME cache before removing clean data	$\mathbb{N}^*$	80	%

2.3.3 Accelerable applications

The SBB is designed to absorb large spikes in the I/O streams, and we focus in this thesis on the optimization of two different scenarios of I/O patterns specifically sensitive to the SBB and particularly sensitive to auto-optimization. To create these benchmarking applications, we use the Open-Source benchmarking application FIO [2]. Similarly to the SRO benchmark, these patterns give a general idea of the I/O distribution within the file, and can have a wide variation in terms of the values of parameters other than the I/O distribution, such as the number of operations, the size of the performed operations and the size of the file.

Scenario description We study the impact of the SBB on two different I/O intensive scenarios, often encountered in scientific applications:

(a) **I/O bandwidth sensitive benchmark**: This type of benchmark represents a checkpoint being written simultaneously by all application's processes which saturates the available RDMA bandwidth. This is representative of a scenario where datanodes are shared between jobs and where the optimizer is forced 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.

(b) **I/O latency sensitive benchmark**: It consists in performing a mix of read and write operations. 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 and the auto-tuner must learn to give enough resources for RDMA connections (RDMA polling threads) and insertion into the RAM cache (workers threads). There is also a significant fraction of reuse on the cache, that can be exploited for performance improvement by setting the RAM cache threshold high enough. In this case, the NVME cache is not used and its parameters are not explored (flash destagers, flash threshold and NVME cache size) as it does not impact the overall performance of this scenario.

Impact of the accelerators on these patterns Using the FIO benchmark [2], we implemented two instances of the scenarios described in the previous section, and ran them with a wide set of different Smart Burst Buffer parameters. From these experiments, we find that the different benchmarks are highly sensitive to their parametrization, as can attest figures 2.7 and 2.8. This complicated landscape motivates the importance of tuning to ensure that the system's performance is maximized, as well as the relevance of black-box optimization to navigate such complex interactions. Another key motivation is the unsuitability of the default parametrization for the two benchmarks, as the optimum parameters give a 37% performance improvement over the default ones for the I/O bandwidth sensitive benchmark and 20% for the I/O latency sensitive benchmark, even when considering the largest possible cache size. More details on these experiments are given in chapter 4.

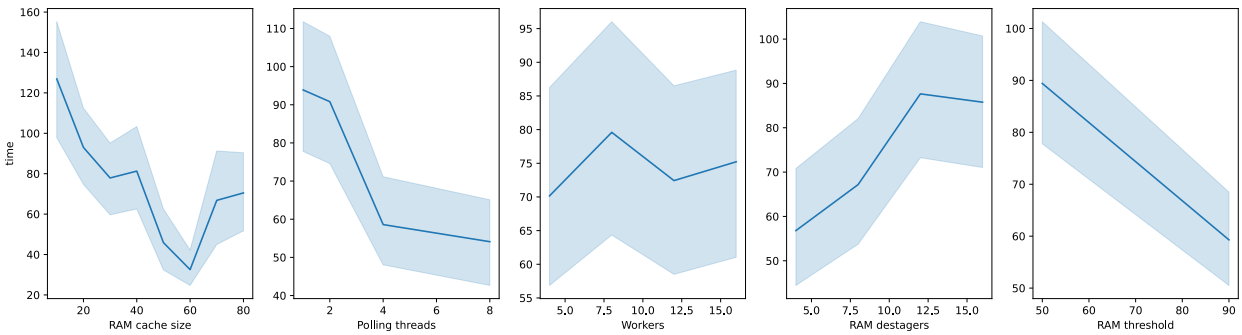


Figure 2.7: Impact of Smart Burst Buffer parameters on I/O bandwidth sensitive benchmark

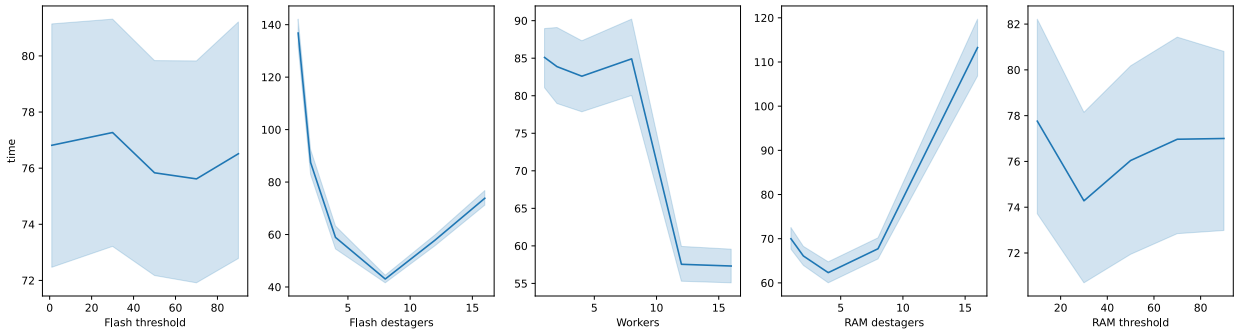


Figure 2.8: Impact of Smart Burst Buffer parameters on I/O latency sensitive benchmark

2.4 Conclusion

In this chapter, we presented the two main tunable systems that are used as use-case to validate our auto-tuning methodology: a smart prefetch strategy called Small Read Optimizer and a burst buffer called Smart Burst Buffer. We described several benchmarking applications used with these accelerators and demonstrated the importance of using them with the right parameters to maximize performance. Different variations of these benchmarks will be used throughout this thesis to validate the different optimization methods.

The next chapter describes in-depth the different methods selected to perform the black-box optimization tuning of these accelerators.

Bibliography

[1] *Atos boosts hpc application efficiency with its new flash accelerator solution*. https://atos.net/en/2019/product-news_2019_02_07/atos-boosts-hpc-application-efficiency-new-flash-accelerator-solution.

[2] *Fio benchmark*. https://fio.readthedocs.io/.

[3] *Infinite memory engine*. https://www.ddn.com/products/ime-flash-native-data-cache/.

[4] *Overview of cray's datawarp*. https://www.nersc.gov/assets/Uploads/dw-overview-overby.pdf.

[5] *Tools to improve your efficiency*. https://atos.net/wp-content/uploads/2018/07/CT_J1103_180616_RY_F_TOOLSTOIMPR_WEB.pdf.

[6] D. A. Reed and J. Dongarra, *Exascale computing and big data*, in Communications of the ACM, vol. 58, 2015, pp. 56–68.

[7] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, *Storage challenges at los alamos national lab*, in 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012, pp. 1–5.

[8] F. Cappello, A. Geist, B. Gropp, L. Kalé, B. Kramer, and M. Snir, *Toward exascale resilience*, in IJHPCA, vol. 23, 2009, pp. 374–388.

[9] B. R. Landsteiner, D. Henseler, D. Petesch, and N. J. Wright, *Architecture and design of cray datawarp*, in Cray User Group '16, 2016.

[10] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, *On the role of burst buffers in leadership-class storage systems*, in IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2012, pp. 1–11.

[11] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, *BurstMem: A high-performance burst buffer system for scientific applications*, in 2014 IEEE International Conference on Big Data (Big Data), IEEE, 2014, pp. 71–79.

Chapter 3

Black-box optimization

Contents

This chapter presents the main principles of black-box optimization and their applications to computers' auto-tuning. As described in chapter 1, auto-tuning has emerged as an adaptive tuning solution which makes no hypothesis on the tuned component and its interaction with its environment. It consists in treating the tuned system as a black-box, deriving insight only from the relationship between the input and the output parameters, as described in figure 3.1 (also represented in figure 1.2). As these methods are oblivious of the tuned system, they do not require any knowledge about the performance function and can be applied to a wide diversity of tunable components.

When dealing with HPC components, the tuned black-box consists in the combination between the component, the application and the execution context. The input parameters of the black-box are the

parameters of the tuned system and the output is a measure of the application's performance, which can for example be the elapsed time of the application or its bandwidth.

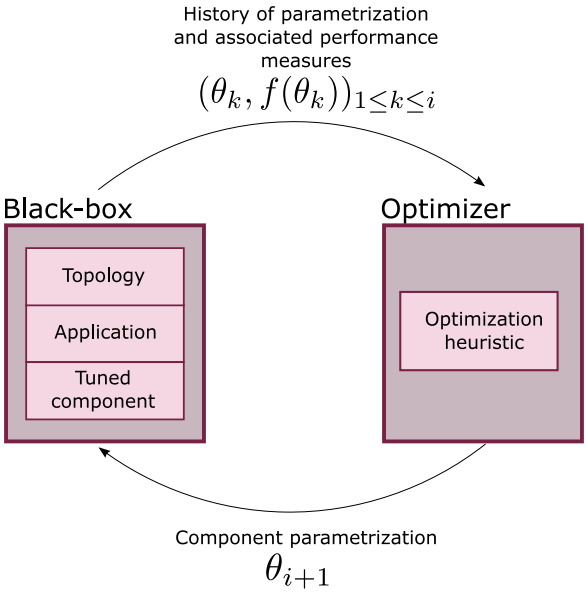


Figure 3.1: Schematic representation of the feedback loop used for auto-tuning of computer systems

3.1 Principles of black-box optimization

Black-box optimization refers to the optimization of a function of unknown properties, most of the time costly to evaluate and which can only be evaluated a limited number of times. As no hypothesis is made on the optimized function, the only information available to the optimizer is the history of the black-box function, as the list of the inputs and the corresponding outputs.

Definition 5: Black-box optimization

Black-box optimization consists in optimizing a function without making any assumption, using only the previous evaluation history and within a limited budget n.

$$\text{Find } \{\theta_i\}_{1 \leq i \leq n} \in \Theta \text{ s.t. } |min(\{f(\theta_i)\}_{1 \leq i \leq n}) - min(f)| \leq \epsilon \tag{3.1}$$

with:

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

The search of the optimum can be divided in two steps:

1. **Initialization step**: it consists in an initial sampling of the parametric space in order to evaluate the performance of starting points. These points are chosen according to the properties of the parametric space, using an *initialization strategy*.
2. **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 an *optimization heuristic*, defined as a higher procedure for searching an optimal solution in a parametric space. This step is iteratively performed until the search satisfies a *convergence criterion*.

A black-box optimization procedure can be described by algorithm 1.

Algorithm 1: Black-box optimization algorithm

Data: Target function f; $\mathcal{F}_i$ the values of the black-box function evaluated at iteration i; $\mathcal{H}$ a heuristic; possible configuration space Θ; IS an initialization strategy; $\mathcal{C}$ a convergence criteria; S_i a sample of selected points at iteration i;θ_i the parametrization selected at iteration i

Result: Optimal parametrization

```
i ← 0
S_0 ← IS(Θ)
while C(S_i, F_i) do
    θ_i ← H(S_i, F_i)
    S_i ← S_i ∪ {θ_i}
    F_i ← F_i ∪ {f(θ)}
    i ← i + 1
```

To be efficient, a black-box optimization heuristic must be a trade-off between exploration and exploitation. It must keep on exploiting zones identified as being good solutions during the initialization or the optimization process, but also explore 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. For each of the heuristic described in this chapter, we will explicit their exploration and exploitation components.

3.2 Initial sampling

The first step of any black-box optimization algorithm, often called *sampling phase* or *design of experiments*, is the selection of the initial parameters to evaluate the black-box function on. The selection of these parameters is very important, as largely discussed in [27] [2] [19] [22] [7], because they give us the very first information about the black-box function.

An acceptable initialization starting plan must respect several constraints, as detailed in [10][26]:

- The **constraint of the parametric space**: they are defined by the nature of the parameters

and the possible values they can be set to.

- The **non-collapse property**: no sampled points should have similar value on any dimension.
- **Space-fill properties**: the experiment plan must maximize a measure of the space-fill, by uniformly spreading over the feasible set.

We will denote the initial sampling plan as $\mathcal{S} = \{\theta_i\}_{1 \leq i \leq n}\ \theta \in \Theta$

3.2.1 Parametric space constraints

The shape of the parametric space is defined by the possible values that can be taken by the parameters and their semantic in the system that has to be optimized. In the case of tuning computer systems, the parametric space is delimited by the physical limits of the system and we manipulate integer discrete values only. We thus work in a discrete space, which is bounded by a minimum and a maximum, set by the physical's system's constraints.

3.2.2 Non-collapse property

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 exactly the same coordinates. This is especially important if there is no insight on the individual effect of each parameter.

Respecting this property does not ensure that the parametric space is homogeneously filled, as we can see from figure 3.2. If we want an homogeneous fill, the repartition of the data points on the grid must be controlled using another constraint that maximizes the spread of the sample on the grid. There is a trade-off between space-fill and non-collapse properties: a design with good space-fill properties is often collapsing (for example, the factorial design is fully collapsible). To obtain a good design in terms of both space-fill and non-collapse, the initialization subset is often found within a reduced class of non-collapsing designs, usually Latin Hypercube Designs [17], on which a selected space-fill measure is maximized.

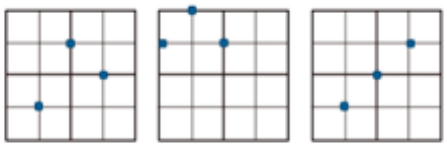


Figure 3.2: Example of designs respecting the non-collapse constraint but not space-fill constraints

3.2.3 Space-filling criteria

Different space-fill measures are available and commonly used to measure the spread of an experiment plan among all feasible ones [27][22]. Such measures are usually divided into two categories: geometrical designs, which rely solely on the geometric properties of the parametric space, and statistical designs which take into account the statistical properties of the black-box function when evaluated for these parameters. Maximizing these criterion are an optimization problem in itself, that can often be solved by traditional optimization techniques, such as brute force if the space is small enough, local improvements [27], column exchange [28], or even by using optimization heuristics, such as simulated annealing [19] and genetic algorithms [4] to find the design with the best value.

Geometrical designs

Geometrical designs rely only on the geometric properties of the parametric space and make no assumption on the behavior of the black-box function.
Non-exhaustively, we can cite:

- **Factorial designs**: factorial designs consist in dividing each dimension i of the parametric space into n_i intervals, and taking the center of each of these cut intervals. Factorial designs are highly uniform [27], but do not respect the non-collapse property. They also require a high number of sampling points to respect uniformity ($n = n_1 \times n_2 \times \cdots \times n_d$), and are not suited for small sampling sets, unless the parametric space is relatively small.

- **Latin Hypercube Designs**: latin hypercube designs (LHD)[17] are among the most represented non-collapsible designs in the literature. They consist in dividing each axis into n equal intervals for each dimension i until the maximum dimension d. This creates n^d identical cell. An LHD is then built by assigning all n initialization points to the center of each of these cells, so that there are no two points with the same coordinate in any dimension. While this makes the design fully non-collapsible, it does not ensure any homogeneity over the parametric space, as can be seen in figure 3.2. Mathematically, when representing a design matrix as a matrix with d columns, corresponding to the dimension of the parametric space, and n row, corresponding to the number of sampled point, where each element is the sampled parametrization, this design is a Latin Hypercube Design if each column of the design matrix is a permutation of {1, 2, . . . , n}.

  Definition 6: Latin Hypercube Design

  A Latin Hybercube Design or Sampling (LHD or LHDS) [7] with n initial starting points and d dimensions is an $n \times d$ matrix, in which each column is a random permutation of $\{1, 2, \ldots, n\}$.

- **Minimax design**: minimax designs consist in making sure that each point of the domain is "close" to a point sampled by the initialization algorithm. The proximity between two parametrization of the parametric space is defined by a metric. This requires that the sampling domain is measurable and that a metric can be defined to measure proximity between data points [27] [7] [14]. If we define a metric M and note $M(\theta, S) = min_{y,\theta}(M(\theta, y))$ *i.e.* $M(\theta, S)$ represents the smallest distance between the data point x and any point of the sampled space S. The set S^* of cardinality n respects the minimax distance if it solves:

  $$\min(\max_{\theta \in \Theta} M(x, S))$$

  which consists in sampling the initial points to minimize the maximal distance between each data point of the space.

- **Maximin design**: On the contrary, the maximin sample wants to make sure that no point of the initial domain are close to each other and that we do not evaluate starting points that are too close to each other [27] [7] [14]. Using the same notations as we did for the minimax design, we can translate this problem mathematically as: $\max(\min_{\theta, y \in \Theta}(M(\theta, y))$

Statistical designs

These designs make statistical assumptions on the response of black-box function and infer from these some constraints on the design matrix. These type of designs make the assumption that the optimized function is a stochastic process X, with mean 0 and a correlation function R. This correlation function R, which describes the relationship between parametrizations across the space, has the property that when two parametrizations θ_i and θ_j are close, their correlation is high, and when they are far away, their correlation is low [27]. The experiment plan $\mathcal{S}$ is selected so that the variable X can be best predicted, by exploiting the information about the correlation function R and fitting a surrogate model $(\hat{X}(\theta))$ to predict the value of the black-box function at parametrization θ. Possible surrogate functions and associated correlation functions are discussed in section 3.3.2.

Among the most popular optimality criterion [27], we can cite :

- **Integrated Mean Square Error (IMSE) optimality** (also called **I-optimality)**[7]: The integral mean square error is defined as:

  $$\int_{\Theta} \text{Var}\hat{X}(\theta)\phi(\theta)d\theta$$

  In the case of a finite design space, ϕ is often set to 1, and solving the optimal design consists in

finding:

$$\min_{|S|=n} \sum_{\theta_i \in \Theta} \text{Var}\hat{X}(\theta_i)$$

which is equivalent to finding the sampling plan which minimizes the total error on each of the predicted points.

- **Entropy criterion** (also called **D-optimality)**[22][25]: for each design set, we can define the correlation matrix of this set by a square matrix

$$R(\mathcal{S}) = [Corr(X(\theta_i), X(\theta_j))]_{1 \leq i,j \leq n}$$

An initialization plan minimizes the expected posterior entropy (which corresponds to the sample points where we have the least information on) if it minimizes [25]:$-\log(\det(R(\mathcal{S})))$

3.3 Optimization heuristics

Optimization heuristics are strategies used to guide the search process when looking for the optimum of a function, most of the time by making little to no assumption on the optimization problem. Their goal is to efficiently explore the search space in order to find near-optimal solutions. As they are not problem-specific, they are suitable for solving a wide range of different use-cases and have been successfully used in various optimization fields. Many heuristics are available in the literature to perform the optimization, and we choose to focus on a subset of possible black-box heuristics. They have been selected by both their efficiency proven in previous works related to ours [6][18][20] and the simplicity of their implementation:

3.3.1 Genetic algorithms

Genetic algorithm is an optimization algorithm which mechanism is inspired from biological evolution. They are classified as stochastic, population-based optimization heuristics. They rely on the idea of making a population evolve thanks to a breeding mechanism between promising solutions which transmits the best characteristics of each parent to the next generation.

Definition and vocabulary

Genetic algorithms consist in making a subset of solution, called population, evolve.

Definition 7: Population (genetic algorithm)

A **population** is the subset of solutions selected by genetic algorithms.

Each individual from this population can be entirely described by their genotype which encodes

the solution into the space of possible parametrizations of the system. Throughout this thesis, we made the choice to consider only genetic algorithms which draw only 2 individuals from the population to perform breeding. They will then generate a single offspring at each generation, evaluated sequentially at each iteration by the tuned system.

Definition 8: Genotype (genetic algorithm)

The **genotype** of an individual among a population of a genetic algorithm is the representation of a solution of an optimization problem in a projected space. This defines a bijective function which associates from each solution its genotype and reciprocally.

A measure of aptitude (called fitness) is associated to each individual and thus to each genotype. It translates how fit is the individual for solving the considered problem.

Definition 9: Fitness function (genetic algorithms)

The **fitness function** is a function which associates to each individual of the population a measurement of its capacity at solving the problem at hand. In general, for optimization problems, it consists in the value of the function to optimize measured on the given solution.

The population is manipulated by operators at each step of its evolution process. They allow the selection of individuals that will be allowed to mate and how the breeding is performed.

Algorithm

The execution workflow of genetic algorithms, represented in figure 3.3 and described in algorithm 2, consists in selecting two combinations of parameters among the set of already tested parametrization, according to a selection process that takes into account the fitness of each parent. These two parameters are then combined to create a new one, using a crossover method. This newly created combination can undergo a mutation, which subtly alters it in order to provide a new combination. The exploitation component is embodied by the selection process, while the exploration happens when the

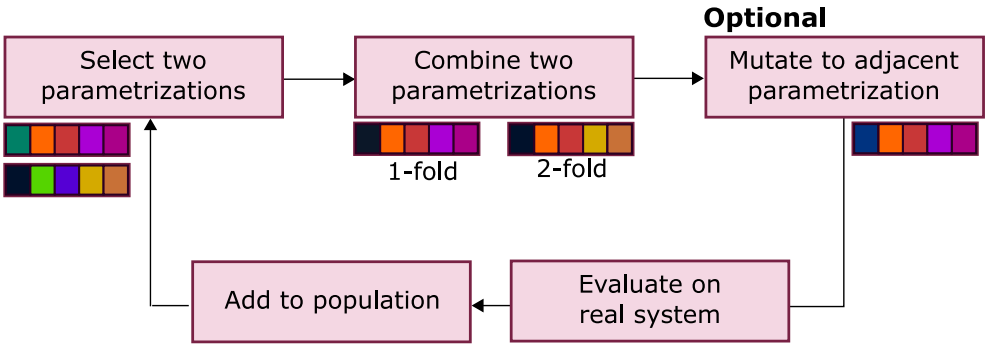


Figure 3.3: Schematic representation of a genetic algorithm

offspring is mutated.

Algorithm 2: Genetic algorithm

Data: $\mathcal{F}_i$ the values of the black-box function evaluated at iteration i; S_i a sample of selected points at iteration i; **selection** a selection function; **crossover** a crossover function; **mutation** a mutation function

Result: A new parametrization to test

```
parent_1, parent_2 ← selection(F_i, S_i)
offspring ← crossover(parent_1, parent_2)
if random then
    offspring ← mutation(offspring)
return offspring
```

Selection

Definition 10: Selection (genetic algorithm)

A **selection** function is a function which selects two individuals in order to breed them.

The goal of the selection step of the genetic algorithm is to select the two fittest parents that will reproduce in order to generate a new solution. Many different methods exist for such selection [23]. We consider and describe the two used in this thesis, selected among the most popular selection methods.

Tournament pick Tournament consists in randomly drawing two subsets of the already tested values of a given size. The individual of each pool with the best fitness is then selected.

Probabilistic/Roulette wheel pick Probabilistic/roulette wheel pick consists in randomly drawing the two parents among the population according to a distribution law proportional to the fitness of the population. This entails that individuals with a higher fitness are statistically more likely to be selected to reproduce. Enforcing *elitism*, so that the individuals with the highest fitness always has a probability

of being selected is also common when picking out the two parents [23].

Reproduction

The reproduction step merges the two individuals selected by the previous step into one. This way, the child created by this breeding process inherits characteristics from both parents.

Definition 11: Reproduction / Crossover (genetic algorithm)

A reproduction or crossover function is a function which merges two individuals into one which inherits characteristics from both the parents.

Most reproduction methods use crossover or a variant. Crossover is the method by which the two parametrizations will merge in order to create a new one. The most common method, as inspired by biology, is to use single-point crossover, which consists into randomly splitting each parametrization in two and concatenating the two parents. Variants of this technique are called *n*-points crossover and consist in cutting the parents into n parts and alternatively concatenating them. Single and double points crossovers are illustrated in figure 3.4.

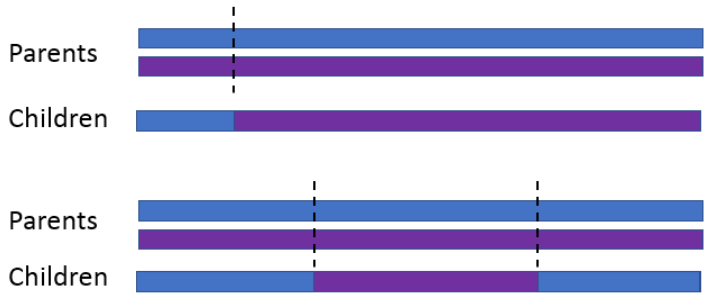


Figure 3.4: Example of representation of single and double point crossovers

Mutation

Definition 12: Mutation (genetic algorithm)

A **mutation** function is a function which alters a genotype to turn it into another one.

Several mutation methods can be considered, but we only consider one in the coming experiments: a random walk across the parametric space, where the considered parametrization is merged randomly into one of its nearest neighbor.

Genotype encoding

The parametric space is transformed using the identity function. We consider the individual being equivalent to a parametrization of the component and the different genetic algorithm processes are

directly applied to the raw parametrization values.

3.3.2 Sequential Model Based Optimization

Sequential Model Based Optimization (SMBO), also called surrogate modeling and Bayesian Optimization, consists in using a **probabilistic model** to approximate the function to optimize and to use this model to select the next data point that will be evaluated by the real function [9] [27] [15] [24]. This model is less costly to evaluate than the true function and thus many data points can be evaluated in order to have a better approximation of the original function.

Definition 13: Model Based Optimization

Model Based Optimization methods construct a regression model, often called surrogate model or response surface model, that predicts performance and then use this model to select the next data point to be evaluated.

Based on this new representation, an **acquisition function** is computed to indicate for each parametrization its potential performance improvement by being evaluated next. This acquisition function must be maximized at each iteration. Choosing the right acquisition function is very important as it describes the importance of each data point in the optimization process, and is responsible for not wasting important computing resources by selecting sub-optimal parametrizations. It also encodes the exploration-exploitation trade-off.

Definition 14: Sequential Model Based Optimization

Sequential model based optimization iterates between fitting a model and evaluating additional data based on this model.

SMBO can be summarized as iterating between (1) fitting a probabilistic model on the known data points (2) selecting the most promising data point using an acquisition function and (3) evaluating the value of the function at the selected data point, as described in algorithm 3.

Algorithm 3: Sequential model based optimization algorithm

Data: $\mathcal{F}_i$ the values of the black-box function evaluated at iteration i; S_i a sample of selected points at iteration i; **regressor** a function to build the surrogate model; **acquisition_function** a function to select the next parametrization to try;
Result: A new parametrization to test

$\mathcal{M} \leftarrow$ **regressor**$(\mathcal{S}_i, \mathcal{F}_i)$
return $\text{argmax}_{\theta \in \Theta}$**acquisition_function**$(\theta, \mathcal{M})$

Figure 3.5 illustrates Bayesian optimization on a 1D function, which is represented as a dashed line. The probabilistic model is represented as the black full line, and the uncertainty around each prediction is represented by the blue area: there is no uncertainty around observations. The value

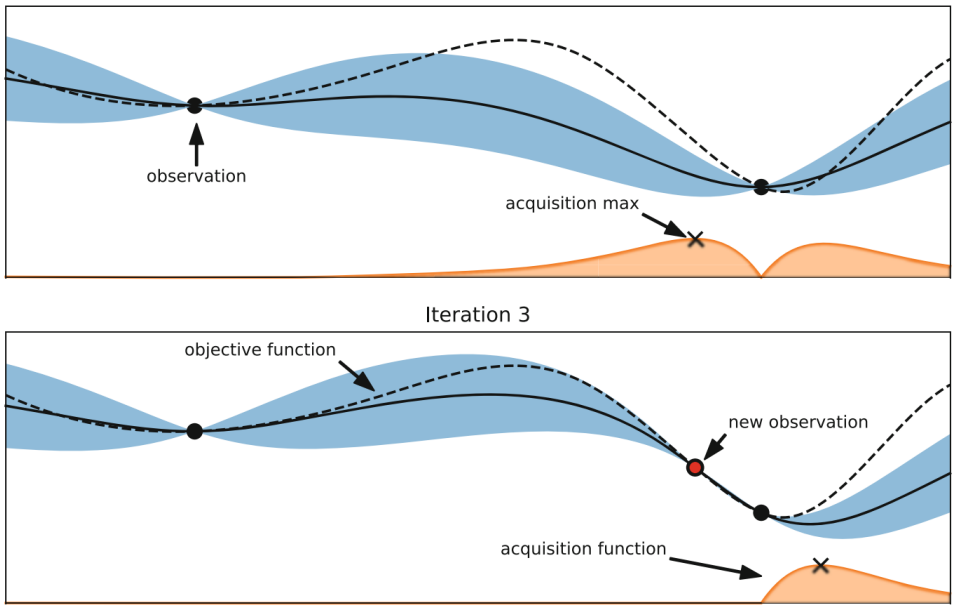


Figure 3.5: Schematic representation of the SMBO algorithm taken from [12]

of the acquisition function is represented by the lower orange tube. In the top figure, the acquisition value is highest for a data point where the predicted function value is low and the predictive uncertainty is high. This data point is selected and the fitted model is refitted to take into account the next observation, as can be seen in the bottom figure.

SMBO requires the selection of two variants among the existing ones: the acquisition function and the surrogate function or response surface model used to represent the performance function.

Definition 15: Acquisition function

An acquisition function u is a function which encodes for each of the parametrization of the parameter grid its potential usefulness for solving the optimization problem.

$$u : \mathbb{P} \longrightarrow \mathbb{R}$$

Probabilistic models

A suitable probabilistic model should be able to give an estimation of the mean and the standard deviation for each possible parametrization. The most popular choice are Gaussian processes[11][12], because they provide smooth uncertainty estimates, a well as a closed-form expression of the predictive distribution.

A Gaussian Process is fully characterized by a mean function $\mu(.)$ at each parametrization, as well as a covariance function $\Sigma(.,.)$ between all of the parametrization of the parametric grid. Mean and

variance prediction at parametrization θ_i are obtained as:

$$\mu(\theta_i) = k_* K^1 y$$

$$\sigma^2(\theta_i) = \Sigma(\theta_i, \theta_i) - k_*^T K^{-1} k_*$$

with k_* denotes the vector of covariances between all previous observations, K is the covariance matrix of all previously evaluated configurations and y are the observed performance values. The selected covariance function $\Sigma(.,.)$ has a strong impact on the performance of the model[12], and we opted for the common choice of a radial-basis function kernel:

$$\Sigma(\theta_i, \theta_j) = exp(d(x_i, x_j))$$

with $d(.,.)$ the euclidean distance.

The other popular model type are tree-based approaches, which are particularly well suited to handle high-dimensional and partially categorical input spaces. In particular, the *sequential model-based algorithm configuration* (SMAC), as presented in [11], uses random forests modified to yield an uncertainty estimate. Another tree-based approach is the Tree Parzen Estimator (TPE) which constructs a density estimate over good and bad instantiations of each hyperparameter.

As Gaussian Processes provide good predictions in low-dimensional numerical input spaces which will be the case of our evaluation problem, we will use them throughout the rest of this thesis.

Acquisition functions

Given the surrogate model, an acquisition function should be able to find the most promising data point.

Using the surrogate as the acquisition function The simplest method is to use the surrogate as the cost function and select its optimum. While very simple, this method does not leverage the statistical information yielded from the probabilistic model. Several methods, such as brute force minimizer, CMA minimizer [13], or L-BFGS [5] exist in order to find the minimum of a function and detailing each of them is out of the scope of this thesis. These methods have been discarded because they have proven to be less efficient [16]. Note that the nature of the parametric space can impact the choice of the minimizing algorithm: a bounded parametric space may for example limit the possible algorithms to use.

Maximum probability of improvement (MPI) In this case, the acquisition function is the probability that the f value for the next point will be lower than the current best value f^*, *i.e.* computing $\mathbb{P}(f(x) \leq f^*)$.

Formula 1: Maximum probability of improvement

Under the assumption that the black-box function can be accurately described by a probabilistic model, the estimated mean and standard error for parameter θ, $\mu(\theta)$ and $\sigma(\theta)$ are available. We can derive that $\forall\theta \in \Theta$ the **Maximum Probability of Improvement** can be defined as:

$$\mathbb{P}(f(\theta) \leq f^*) = \Phi(\frac{f^* - \hat{\mu}(\theta)}{\hat{\sigma}(\theta)}) \tag{3.2}$$

Φ being the normal cumulative function.

This method encodes the probability of the parametrization giving a reward compared to the current optimum, but does not take its potential value into account. As the search region under a given parameter θ becomes more and more known, $\sigma(\theta)$ gets smaller and so does $MPI(\theta)$. The algorithm then switches to a different search region.

Expected improvement (EI) This method both encodes exploration (exploring spaces with a high variance) and exploitation (exploring spaces with a low mean). Not only it takes into account whether or not the next point will induce a smaller value for f but it also considers the gain from the switch. Unlike MPI which only looks for a smaller value, EI takes into account the difference between the old and the new data point.

For each point of the parametric space, we can define the improvement as:

$$I(\theta) = \begin{cases} f^* - f(\theta) & f(\theta) < f^* \\ 0 & f(\theta) \geq f^* \end{cases} \tag{3.3}$$

Which can also be written as $I(\theta) = max(f^* - f(\theta), 0)$.

In the case that f is approximated by a probabilistic process, I is a random variable and its expectation over each possible data point can be computed. The expected improvement criterion consists in finding the parametrization which maximizes the expectation of the random variable I.

Formula 2: Closed form of the expected improvement

Under the assumption that the black-box function can be accurately described by a probabilistic model, the estimated mean and standard error for parameter x, $\mu(x)$ and $\sigma(x)$ are available. We can derive that $\forall x \in \mathbb{P}$ the **Expected Improvement** can be defined as from [15]:

$$EI(x) = \mathbb{E}(I(x)) = (f^* - \mu(x)) \times \Phi(\frac{f^* - \mu(x)}{\sigma(x)}) + \sigma(x)\phi(\frac{f^* - \mu(x)}{\sigma(x)})$$

The closed form of the EI can be easily optimized using any optimization algorithm.

This acquisition function encodes the exploitation-exploration trade-off by exploiting points with a low mean (best estimated function value) and explore points with a high variance (high uncertainty).

3.3.3 Simulated annealing

The simulated annealing heuristic [21] 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: this introduces the notion of the system's "temperature" (hence the analogy with metal annealing).

Definition 16: Simulated annealing

Simulated Annealing is a black-box optimization heuristic inspired by statistical mechanics in thermodynamics with the statistical ensemble of the probability distribution over all possible states of a system described by a Markov chain, where its stationary distribution converts to an optimal distribution during a cooling process after reaching the equilibrium.

The temperature is a value that decreases overtime according to a cooling schedule which defines how the temperature decreases per iteration. As the temperature lowers, the probability of moving upward (*i.e.* accepting a worse solution) decreases until it reaches 0 and the system cannot move anymore.

Description of the algorithm

At each time step, the algorithm randomly selects a solution neighboring the current one, measures its quality, and then makes a choice according to the new value. If the new value is better than the current value, it is automatically accepted. If not, a probability of acceptance is computed. If this probability is lower than a number randomly drawn from the uniform distribution, the new value is accepted even though it is worse than the current state. This makes sure that the algorithm can move

out of a local minimum if it gets stuck in one. The temperature is then reduced one step, according to a cooling schedule. This ensures that as the temperature cools down, the probability of accepting a solution getting worse than the current one lowers. The exploration component of the algorithm decreases so that it can focus on exploitation. At the end of the algorithm, as the temperature looms close to zero, the probability of acceptance of a worse solution draws to 0 and no solution worse than the current one can be accepted. In some variant, the algorithm can spend a given number of steps at the same temperature before the cooling schedule is applied. The algorithm workflow is given in algorithm 4 and illustrated in figure 3.6.

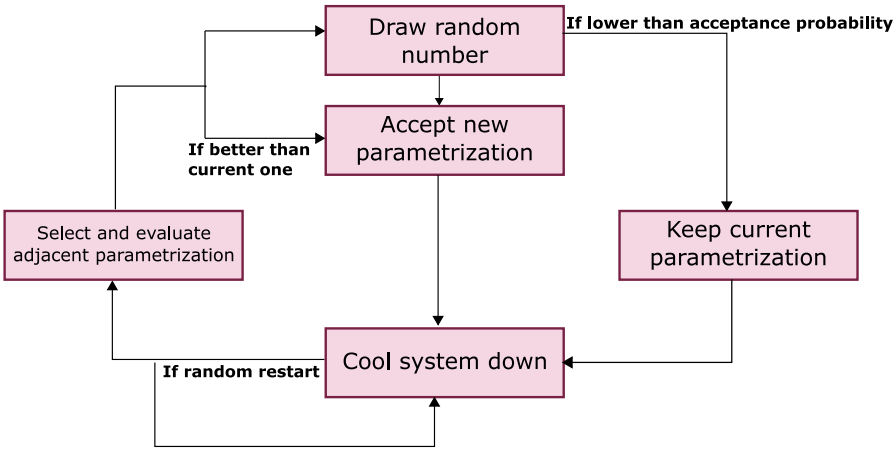


Figure 3.6: Schematic representation of the simulated annealing algorithm

From this description, we see that three functions need to be defined in order to implement the algorithm:

- **The acceptance probability function**: this function decides the probability of accepting a value lower than the current state. As it is a probability function, it needs to follow the properties common to all probability laws. It also has to have the following property

$$lim_{T \to 0} \mathbb{P}(P(y_{new}, y_{old}, T)) = 1$$

so that the probability of accepting worse solution goes down as the temperature reduces.

Definition 17: Probability of acceptance (simulated annealing)

The **probability of acceptance** is a function $f : (\mathbb{R}, \mathbb{R}, \mathbb{R}) \longrightarrow [0, 1]$ which computes the probability of accepting a value worse than the current state, given the current, the new value and the system's temperature. If this value is higher than a certain random threshold, a solution worse than the current one is accepted.

- **The cooling schedule of cooling function**: this function computes the temperature at iteration k when given the initial temperature.

> Definition 18: Cooling schedule (simulated annealing)
>
> The **cooling schedule** is a decreasing function $f : \mathbb{N} \longrightarrow \mathbb{R}$ which computes the value of the temperature at iteration k.

- **The neighboring function**: given a data point in the sample space, this function returns a point that can be considered as a neighbor of this data point.

> Definition 19: Neighboring function (simulated annealing)
>
> The **neigbhoring function** is a function $f : \Theta \longrightarrow \Theta$ which, given the currently considered parametrization, returns a parametrization adjacent on the parametric grid.

Algorithm 4: Simulated annealing

Data: θ_i parametrization at iteration i; θ_{i+1} parametrization at iteration $i+1$; f_i value of performance function at parametrization θ_i; f_{i+1} value of performance function at parametrization θ_{i+1}; T_i the system's temperature; probability_acceptance the probability of accepting a function worse than the current one; neighbor a function returning a neighboring parametrization; **cooling_schedule** the cooling schedule

Result: A new parametrization to test

```
x_{i+1} ← neighbor(x_i)
f_i ← f(x_i)
f_{i+1} ← f(x_{i+1})
if f_{i+1} < f_i then
    return neighbor(x_{i+1})
else
    L ← random(0, 1)
    P ← probability_acceptance(f_i, f_{i+1}, T_i)
    if P > L then
        return neighbor(x_{i+1})
    else
        return x_i
T_{i+1} ← cooling_schedule(T_i)
```

Probability of acceptance

In the initial description of the optimization procedure [21], the probability of acceptance function, which returns the probability to accept the new parametrization, is defined as:

$$f(y_{new}, y_{old}, T_k) = \mathbb{P}(\text{accept} y_{new}) = \begin{cases} exp(\frac{y_{new} - y_{old}}{T_k}) & y_{new} \geq y_{old} \\ 1 & y_{new} \geq y_{old} \end{cases}$$

with y_{new} the value of the black-box function at the newly tested parametrization, y_{old} the value of the black-box function at the current parametrization, and T_k the temperature of the system at step k. The original authors justified this formula with an analogy to the energy transitions of a physical system.

Defining neighboring values

Any metric can be used to compute the neighborhood of a parameter on the grid. In our practical experiments, we use a random walk on the parameter grid, that can move in any direction. This random walk is illustrated in figure 3.7, where the currently selected parametrization is represented in blue and the possible parametrization that could be selected by the algorithm, using a random Bernoulli law in all three directions (move up one, go back one, or stay in the same place).

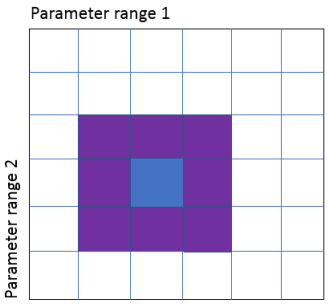


Figure 3.7: Possible selected parametrizations when using a random walk

Cooling schedules

Any functions which decreases towards zeros as the number of iteration rises can be used as a cooling schedule. By denoting T_0 the initial temperature, k the iteration round, T_k the temperature at round k and α the cooling factor, we test:

- **Exponential schedule**: It is the original cooling schedule suggested by the authors in [21], where they suggest a cooling factor of 0.99: $T_k = \alpha^k \times T_0$
- **Multiplicative linear schedule**: $T_k = \frac{1=T_0}{1+\alpha * k}$ [3]
- **Logarithmic schedule**: $T_k = \frac{T_0}{1+\alpha log(1+k)}$ [1]

The choice of the cooling schedule impacts the rate at which the system stabilizes itself around a solution. The slower the schedule, the longer the system will be able to switch to non-improving solutions, at the risk of spending its iteration budget on suboptimal space and thus being penalized because of its high exploration cost. A representation of each cooling schedule is provided in figure 3.8.

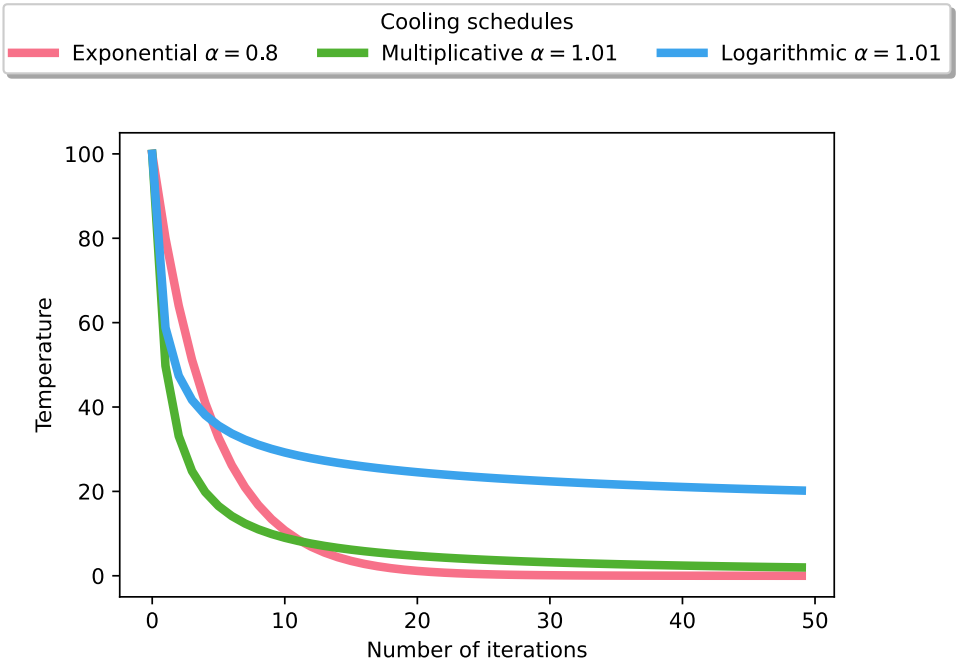


Figure 3.8: Evolution of the temperature for different cooling schedules, $T_0 = 100$

Restarting the algorithm

As the temperature decreases, the simulated annealing algorithm can get stuck in a local optimum and not have enough energy to get out of it. This issue can be addressed by providing the algorithm with a restart mechanism, which sets the temperature back to its initial value. Although several restart criteria have been described and tested in the literature, we experimented with random restarts, which swaps the temperature with the initial value according to a Bernoulli law drawn at each iteration, and a threshold restart, which restarts the system when the energy has gone under a given value. The number of possible restarts is subjected to a budget that cannot be exceeded.

3.4 Convergence criteria

When running the optimization algorithm, the easiest stop criterion is based either on a budget of possible steps (also called exhaustion based) or on a time-out based on the maximum elapsed time for the algorithm. Once the iteration budget has been spent, the algorithm stops and returns the best found parametrization. However, while very simple to implement, this criterion can be inefficient, as it has no adaptive quality on the tuned system. We present other possible criteria often found in the literature [29].

Exhaustion based criteria Exhaustion based criteria are criteria based on the number of allowed iterations performed by the heuristic. Once all of the possible iterations have been tried by the algorithm,

the algorithm stops and the parametrization which returned the best corresponding performance measure is kept. They are the most popular in the black-box optimization literature [29] because of their simplicity of implementation and the control they give over the optimization process. However, they can be a waste of resource because they can:

- Stop the algorithm while the maximum optimization potential has not been reached, thus not finding the optimal parametrization. The algorithm either has to be started from scratch or can resume, depending on the practical auto-tuning implementation.
- Keep the algorithm running even though the maximum potential of optimization has already been reached. This is a waste of time and resources, as the algorithm runs aimlessly without providing any improvement.

Exhaustion-based criteria thus do not provide much flexibility in the optimization process and do not have any adaptive qualities to the behavior of the system. Their main strength is their simplicity of implementation and the time guarantees given to the user. Because of this, they are a good choice when there is a limited budget of time or of application runs a user is allowed to launch on a cluster.

Referenced based criteria This consists in stopping the algorithm once a value, acting as a reference, has been reached. In real-life, this is impractical, as the improvement potential of the application is unknown and fixing a set value would be too limiting. However, it is used for benchmarking purpose, in order to evaluate convergence speeds of algorithms. A variation of referenced based criteria, specifically geared towards the optimization of computer systems, consists in taking as a reference a certain percentage of the default parametrization. In this case, the algorithm stops once the found parametrization returns a performance measure which exceeds a certain percentage of the performance of the default parametrization.

Improvement based criteria They consist in stopping the optimization process if it does not bring any improvement over a given number of iterations. Depending on the target behavior, the improvement can either be measured globally as the average of the evaluated values or locally as the change in optimum values.

- *Best improvement*: Improvement of the best objective function value is below a threshold t for a number of iterations g
- *Average improvement*: Improvement of the average objective function value is below a threshold t for a number of iterations g
- *Median improvement*: Improvement of the median objective function value is below a threshold t for a number of iterations g

Movement based criteria Movement based criteria consider the movement of the parametric grid as a criteria to stop the optimization. Two variation of the criteria can be designed:

- *Count based*: The optimization algorithm is stopped once there is less than t different parametrization evaluated over a number of iterations g.
- *Distance based*: The optimization algorithm is stopped once the distance between each parametrization goes below a certain threshold t for a number of iterations g.

Combination of several criteria When used in practice, all of the other criteria are usually combined with an exhaustion-based criterion, which ensures a time limit on the optimization process. This way, the user has the guarantee that the optimization process will not exceed a certain budget, but also knows that the optimization process will stop earlier if there is no improvement, which will save resources.

3.5 Evaluation of optimization quality

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 computer systems in an offline and online settings, as described in section 1.3.
To account for the possible constraints associated to online tuning, we define six novel metrics in table 3.1 to rank the heuristics.

Metric name	Code	Description
Success	S	Whether or not the optimal parametrization was selected
5% of the optimal	5%Opt	Number of iterations required to reach a parametrization which is within 5% of the optimal point
Distance between the optimum and the correct value	DistOpt	The distance between the selected optimum and the true optimum
Distance between the 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
Exploration cost	EC	The summed difference of regressions (i.e. the application performing worse than it has in the past)
Elapsed time	ET	The computation time

Table 3.1: Description of the metrics used to compare the different 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. Online tuning indeed

requires consistency and stability in the results. An auto-tuner used online 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 [8]. 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, for example 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.

To average for the possible effects of the random components of the different algorithms (the initialization strategy, the selection of nearest neighbors, the number of mutations, the number of restarts . . .), the metrics must be computed on several instances of each heuristic and their different variants, to give some statistically significant results for comparison.

3.6 Conclusion

In conclusion, we presented in this chapter the principles of black-box optimization. We describe the most common initialization plans, as well as three of the most commonly encountered heuristics (Genetic Algorithms, SMBO and Simulated Annealing). For each of these heuristics, we introduce their different hyperparameters and possible variations, that are used throughout this thesis. Different stop criteria commonly found in the literature are also provided. We also define some original evaluation metrics to quantify the performance of a black-box optimization heuristic on an online and offline tuning context.
The next chapter provides an analysis of the performance of each of these heuristics for tuning the two I/O accelerators described in chapter 4, using the evaluation criteria discussed above. We also study the impact of each hyperparameter on the heuristics behavior.

Bibliography

[1] A. Albrecht and C.-K. Wong, *On logarithmic simulated annealing*, in Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, eds., 2000, pp. 301–314.

[2] A. S. Andy Keane, *Engineering Design via Surrogate Modelling*, John Wiley Sons, Ltd, 2008, ch. 1, pp. 1–31.

[3] M. ATIQULLAH, *An efficient simple cooling schedule for simulated annealing*, 2004, pp. 396–404.

[4] S. BATES, J. SIENZ, AND V. TOROPOV, *Formulation of the optimal latin hypercube design of experiments using a permutation genetic algorithm*, in 45th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics Materials Conference, vol. 2011, 2004.

[5] R. BYRD, P. LU, J. NOCEDAL, AND C. ZHU, *A limited memory algorithm for bound constrained optimization*, SIAM J. Sci. Comput., 16 (1995), pp. 1190–1208.

[6] D. DUNLOP, S. VARRETTE, AND P. BOUVRY, *On the use of a genetic algorithm in high performance computer benchmark tuning*, in International Symposium on Performance Evaluation of Computer and Telecommunication Systems, 2008, pp. 105–113.

[7] K. T. FANG, R. LI, AND A. SUDJIANTO, *Design and Modeling for Computer Experiments (Computer Science & Data Analysis)*, Chapman & Hall/CRC, 2005.

[8] Y. GUR, D. YANG, F. STALSCHUS, AND B. REINWALD, *Adaptive multi-model reinforcement learning for online database tuning*, in 24th International Conference on Extending Database Technology, 2021.

[9] H. H. HOOS, *Automated Algorithm Configuration and Parameter Tuning*, 2012, pp. 37–71.

[10] B. HUSSLAGE, G. RENNEN, E. DAM, AND D. DEN HERTOG, *Space-filling latin hypercube designs for computer experiments*, in Optimization and Engineering, vol. 12, 2006, pp. 611–630.

[11] F. HUTTER, H. H. HOOS, AND K. LEYTON-BROWN, *Sequential model-based optimization for general algorithm configuration*, in Learning and Intelligent Optimization, C. A. C. Coello, ed., 2011, pp. 507–523.

[12] F. HUTTER, L. KOTTHOFF, AND J. VANSCHOREN, eds., *Automated Machine Learning: Methods, Systems, Challenges*, The Springer Series on Challenges in Machine Learning, Springer International Publishing, 2019.

[13] C. IGEL, N. HANSEN, AND S. ROTH, *Covariance Matrix Adaptation for Multi-objective Optimization*, in Evolutionary Computation, vol. 15, 2007, pp. 1–28.

[14] M. JOHNSON, L. MOORE, AND D. YLVISAKER, *Minimax and maximin distance designs*, in Journal of Statistical Planning and Inference, vol. 26, 1990, pp. 131–148.

[15] D. JONES, M. SCHONLAU, AND W. WELCH, *Efficient global optimization of expensive black-box functions*, in Journal of Global Optimization, vol. 13, 1998, pp. 455–492.

[16] M. N. L. VINCENT AND G. GORET, *Self-optimization strategy for io accelerator parameterization*, in International Conference on High Performance Computing, 2018, pp. 157 – 170.

[17] M. MCKAY, R. BECKMAN, AND W. CONOVER, *A comparison of three methods for selecting values of input variables in the analysis of output from a computer code*, in Technometrics, vol. 21, 1979, pp. 239–245.

[18] T. MIYAZAKI, I. SATO, AND N. SHIMIZU, *Bayesian optimization of hpc systems for energy efficiency*, in High Performance Computing, R. Yokota, M. Weiland, D. Keyes, and C. Trinitis, eds., Springer International Publishing, 2018, pp. 44 – 62.

[19] M. D. MORRIS AND T. J. MITCHELL, *Exploratory designs for computational experiments*, in Journal of Statistical Planning and Inference, vol. 43, 1995, pp. 381–402.

[20] V. PLUGARU, F. GEORGATOS, S. VARRETTE, AND P. BOUVRY, *Performance tuning of applications for hpc systems employing simulated annealing optimization*, 2014.

[21] K. S., G. C. D., AND V. M. P., *Optimization by simulated annealing*, SCIENCE, (1983).

[22] J. SACKS, W. J. WELCH, T. J. MITCHELL, AND H. P. WYNN, *Design and Analysis of Computer Experiments*, in Statistical Science, vol. 4, Institute of Mathematical Statistics, 1989, pp. 409 – 423.

[23] N. SAINI, *Review of selection methods in genetic algorithms*, in International Journal Of Engineering And Computer Science, 2017.

[24] B. SHAHRIARI, K. SWERSKY, Z. WANG, R. P. ADAMS, AND N. DE FREITAS, *Taking the human out of the loop: A review of bayesian optimization*, in Proceedings of the IEEE, vol. 104, 2016, pp. 148–175.

[25] M. C. SHEWRY AND H. P. WYNN, *Maximum entropy sampling*, in Journal of Applied Statistics, vol. 14, Taylor Francis, 1987, pp. 165–170.

[26] E. STINSTRA, D. DEN HERTOG, P. STEHOUWER, AND A. VESTJENS, *Constrained maximin designs for computer experiments*, in Journal of Public Economics - J PUBLIC ECON, vol. 45, 2003, pp. 340–346.

[27] K. K. VU, C. D'AMBROSIO, Y. HAMADI, AND L. LIBERTI, *Surrogate-based methods for black-box optimization*, in International Transactions in Operational Research, vol. 24, 2017, pp. 393–424.

[28] K. YE, W. LI, AND A. SUDJIANTO, *Algorithmic construction of symmetric latin hypercube designs*, in Journal of Statistical Planning and Inference, vol. 90, 2000, pp. 145–159.

[29] K. ZIELINSKI, D. PETERS, AND R. LAUR, *Stopping criteria for single-objective optimization*, 2005.

Chapter 4

Auto-tuning of I/O accelerators: a comparative study

Contents

In chapter 3, we presented three different black-box optimization heuristics: Sequential Model Based Optimization, genetic algorithms, and simulated annealing. The relevance and efficiency of each heuristic can vary significantly according to the tuning context [10] [5] [6]. To prove the efficiency of black-box optimization for tuning the two I/O accelerators described in chapter 2, as well

as select the most relevant heuristic for the auto-tuner presented in chapter 1, we perform in this chapter a comparative study of the different heuristics, with a special attention to the requirements specific to the constraints of offline and online tuning, as highlighted in section 1.3.
In addition, the optimization potential of each heuristics hyper-parameters and variants is examined by conducting an experimental study on several different benchmark applications with the two I/O accelerators. Each variant is tested and compared, and then ranked according to its performance for online and offline tuning. Given the scores of each heuristic and its variants, we provide a comprehensive comparison and analysis of the behavior of each heuristic for each benchmark and accelerator. This data is then used to discuss and select the most appropriate for our tuner.

4.1 Selected benchmarks for tuning experimentation

To thoroughly show the relevance of black-box optimization for tuning, we have to perform the tuning on applications that are representative of scientific applications. They need to be sensitive to the accelerators and their parameters and to provide a diversified optimization landscape to show the adaptability of our solution. They also need to be varied in their I/O behavior as well as their hardware to demonstrate the advantages of the flexibility of black-box optimization.

We have selected five I/O benchmarking applications that fit this description and are representative of some I/O patterns encountered in scientific HPC applications. We have made the choice to use benchmarking applications instead of real HPC applications to reduce experiment duration and collect more data, as an exhaustive grid search for a real HPC applications would take months to run. Another reason for the selection of benchmarks is that they allow to focus only on the time spent by the I/O intensive phase of the application, which is the only one optimizable in this case. Because of this, we can use the whole execution time of the application as the optimization target, rather than the time spent doing I/O which is harder to compute and requires a monitoring system.

The chosen benchmarks are variations of those presented in chapter 2. For the Smart Burst Buffer (SBB), they correspond to an implementation of the two applications specifically designed for this accelerator that are described in section 2.3.3. For the Small Read Optimizer (SRO), we have selected three among those described in section 2.2.3. These benchmarks fulfill our required criteria, as they have good potential for acceleration by the accelerators and also a strong sensitivity to the parameters. They also present very diverse parametrization landscape and this diversity allows to highlight the adaptability of black-box optimization.

In the case of the Small Read Optimizer, two different storage back-ends are tested, Network File

System (NFS) [8] and Lustre [2]. This results in a total of a eight experiments. The number of parameters tested per experiment and its approximated duration is described in table 4.1. Through the remainder of this chapter, we identify an application by its *Application ID* as available in this table, and, when relevant, we identify the used backend by suffixing it to the application ID. For instance, the data collected by running the application `SRO.1` with a NFS backend will be designated as `SRO.1.NFS`.

Table 4.1: 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 (≈ 87 hours)
	SBB.2	Lustre	1024	76135 (≈ 21 hours)
Small Read Optimizer	SRO.1	Lustre	1920	113380 (≈ 31 hours)
	SRO.1	NFS	1920	116078 (≈ 32 hours)
	SRO.2	Lustre	1920	40065 (≈ 11 hours)
	SRO.2	NFS	1920	66676 (≈ 18 hours)
	SRO.3	Lustre	1920	24886 (≈ 6 hours)
	SRO.3	NFS	1920	37331 (≈ 10 hours)
Total	8 experiments	2 backends	16684	790890 (≈ 219 hours)

4.1.1 SBB benchmarks

The benchmark applications selected for the Smart Burst Buffer are implementations of those described in section 2.3.3 in chapter 2. Both experiments use the open source I/O benchmark fio[1] to generate the I/O workloads. The fio configuration of the two applications is described in table 4.2.

Table 4.2: 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: implementation of the I/O bandwidth sensitive benchmark**

   The benchmark performs sequential writes generated by 8 processes, by chunks of 10 MB, for a total of 200GB of total I/O volume. 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 representative setup as datanodes are usually shared between jobs, which forces the tuner to find trade-off between the resources 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: implementation of the I/O latency sensitive benchmark**:

   It consists in performing random read-write I/Os, 75% reads and 25% writes generated by 32 processes, by chunk of 4k, 40GB of total I/O volume. 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 storage to hold the data into the datanode's cache.

The tested values for each parametric dimension of the SBB parametric grid are detailed in table 4.3. They were selected according to the data nodes physical characteristics.

Table 4.3: 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%

4.1.2 SRO benchmarks

The SRO is designed to speed-up pseudo-random accesses made by HPC applications. We use a variation of the three accelerable I/O behaviors described in section 2.2.3. A summary of the parametrization of the benchmark for the three tested patterns is described in table 4.4.

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

Table 4.4: 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	$\mathcal{U}(50k, 500M)$	100	$\mathcal{U}(100, 500)$	0

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 4.5.

Table 4.5: 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 4.4 perform only small random read operations, as they are the ones targeted by the accelerator. The I/O patterns of these three benchmarks are often found in industrial applications that we studied, making them a representative choice for our study. They also can be optimized with SRO and 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 a large number 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 number of operations 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. Because of this, we can collect many different landscapes for the different optimization heuristics and provide significant results across several applications.

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, the prefetch strategy must adapt significantly, which introduces changes to the shape of the parametrization landscape and introduce diversity in the experiment. Experiment file sizes have been set to 100GB for the Lustre backend and 20GB for the NFS backend.

To summarize the experiment, we validate black-box optimization on 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.

4.2 Experiment plan

4.2.1 Tested heuristics

We compare the performance of each of the heuristic described in chapter 3, as well as their different hyperparametrization:

(a) **Sequential Model Based Optimization** (see section 3.3.2):
We use Gaussian Process Regression to perform the regression of the function and compare the acquisition function *Maximum Probability of Improvement* (MPI) and *Expected Improvement* (EI). 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.

(b) **Simulated annealing** (see section 3.3.3):
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. We compare the multiplicative and logarithmic schedule, with a cool-down 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 systems temperature back to its original value and resetting it once the systems energy has gone under a given threshold.

(c) **Genetic algorithms** (see section 3.3.1):
Two methods are 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. Crossovers were made using single and double point crossovers when the size of the parametric space allowed it. 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 number of tested hyperparametrization per heuristic is summarized in table 4.6

Table 4.6: Number of tested hyperparametrization per heuristic

Heuristic name	**Number of hyperparametrization**
Sequential Model Based Optimization	2
Simulated Annealing	4
Genetic Algorithms	4
Total	8

4.2.2 Evaluation criteria

To test, validate and compare the relevance of each heuristic and its hyperparametrization, for each benchmark, we compute the values of the metrics available in table 3.1 of chapter 3. To compute them on realistic trajectories, we need to simulate the auto-tuning loop 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 1.2 of chapter 1 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.

For each of the hyperparametrization 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. Using a random sampler as the baseline for comparison has been highlighted as necessary for meaningful comparisons in several studies, as it has given some surprisingly good results in similar studies in adjacent fields [9] [4] [7] [3].

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 3.3. The number of runs was selected as a trade-off between the time needed to run an experiment and the concern for results statistical significance.

4.2.3 Benchmarking environment

The experimentation datasets were built using the exhaustive search heuristic of the Smart HPC Application MANager (SHAMan), dicussed in depth in chapter 6. 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 and noise. 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 of 40TB.

4.3 Analysis of the optimization trajectories

After generating the optimization trajectories, we discuss the different heuristics and compare them based on their score. We have organized the results of the analysis as follows. We begin by discussing the quality of the optimization, *i.e.* how close the heuristic comes to finding the optimal parametrization. We then present the stability of the optimization trajectory and comment on the suitability of each heuristic for online tuning. We then discuss the trade-off between this stability and the quality of the optimization. We also provide and discuss the values of the other available metrics. We give as well as a description of the consistency of the optimizer in terms of the best found parametrization. We finish the analysis by exploring the impact of each hyperparameters on the trajectories.

4.3.1 Quality of the optimization

Table 4.7a 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 Sequential Model Based Optimization with 3.33% and Simulated Annealing shows less optimization power with 5.50% average distance. As can be seen in figure 4.1, the best optimizer between Sequential Model Based Optimization and Genetic Algorithm depends on the optimized application.

Sequential Model Based Optimization tend to be better at optimizing SRO but Genetic Algorithms are better for SBB based datasets. This is mostly due to the smoother 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 Sequential Model Based Optimization. Because the SRO datasets have a lot more irregularities in the effect of the parameters and a less continuous landscape, Sequential Model Based Optimization perform better because they are not constrained to locality as Genetic Algorithms are. Overall, Genetic Algorithms and Sequential Model Based Optimization have close optimization properties, with Genetic Algorithms performing better both in mean (3.20% for Genetic Algorithms against 3.33% for Sequential Model Based Optimization) but Sequential Model Based Op-

Table 4.7: 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
SMBO	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
SMBO	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
SMBO	EI	3.49	28.74	3.33	40.15	54.88	95.50	0.18
SA	Threshold - Multiplicative	1.19	11.28	88.91	120.38	111.07	10.43	0.21

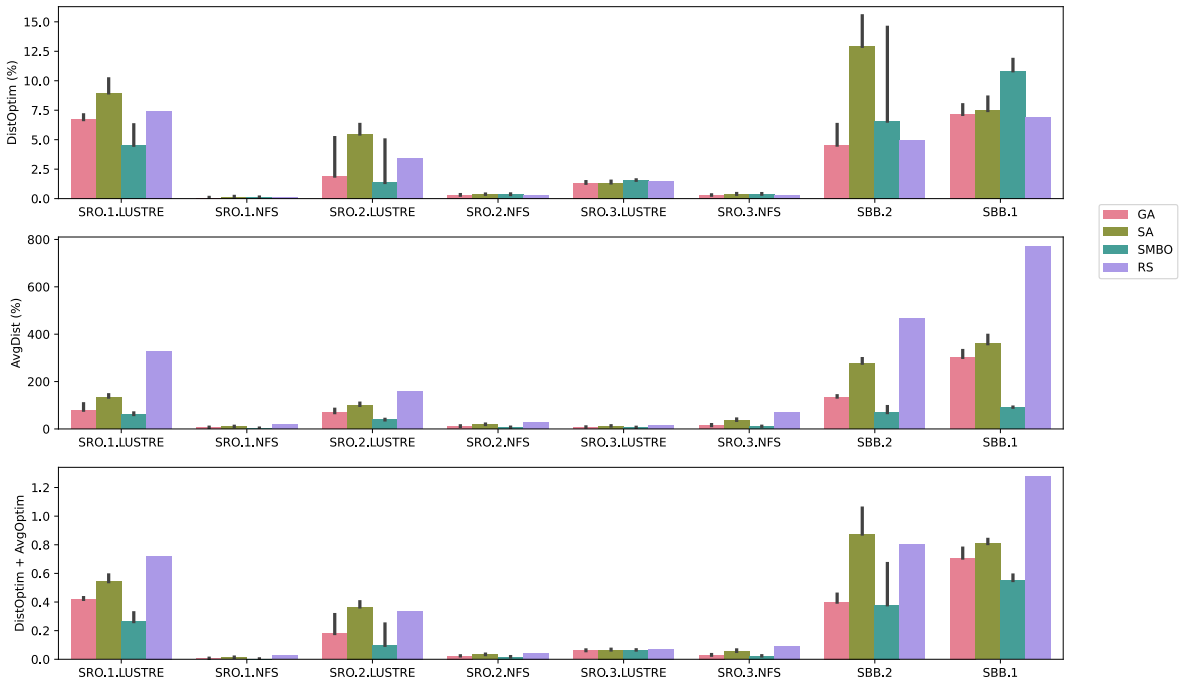


Figure 4.1: Best values for DistOptim, AvgDist and the sum of both for every heuristic

timization performing best in median (1.73% for Genetic Algorithms and 1.48% for Sequential Model Based Optimization).

The results given by using either of these two algorithms confirm the strong optimization and adaptation capability of black-box optimization, as it provides solutions close to the ground truth and

the optimum for every diverse dataset.

4.3.2 Stability of the optimization

The results concerning the stability of the optimization are reported in table 4.7b and in figure 4.1. 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 where stability matters and it reduces the performance of using Random Sampling for optimization. When taking solely into account the AvgDist metric, Sequential Model Based Optimization perform better than Genetic Algorithms (40.15% for the former and 82.83% for the latter) when using MPI, at the cost of a reduced optimization efficiency (3.49% for the former and 2.63% for the former). They both outperform Simulated Annealing in stability (128.03%) and distance to the optimum (5.34%). The stability of Sequential Model Based Optimization 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, Sequential Model Based Optimization offer the most stable behavior for every dataset but the SBB.2 one where it performs significantly worse than the Genetic Algorithms.

4.3.3 Considering both properties

Considering the sum of the normalized variables *DistOptim* and *AvgDist* in table 4.7c allows to rank the heuristics according to both their optimization quality and their stability. Sequential Model Based Optimization provide the best value for this score (0.18), closely followed by Genetic Algorithms (0.19) and Simulated Annealing (0.21).

In conclusion, Sequential Model Based Optimization, parametrized using Expected Improvement gives us some strong convergence properties as well as some guarantees in stability. If considering only these two properties, Sequential Model Based Optimization are the best candidate for tuning our target accelerators.

4.3.4 Elapsed time and convergence speed

When looking at the elapsed time (metric ET), Sequential Model Based Optimization take more time to compute the next parametrization (around 95 seconds on average) compared to Genetic Algorithms and Simulated Annealing (around 10 seconds on average), as can be seen in table 4.7. This is completely aligned with the complexity of the algorithms. It is also consistent with the consumed resources.

When looking at table 4.7, success rates stay very low (below 3.5% for the metric S) 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, the convergence rate as the number of iterations required to come as close as 5% to the ground truth optimum must be taken into account.
When looking at the heuristics with the best trade-off between *AvgDist* and *DistOpt*, Genetic Algorithms requires on average 15.44 steps to be at 5% of the optimum, while Sequential Model Based Optimization have a slower convergence with 28.74 steps. Simulated Annealing has a slower convergence rate with 88.91 required steps to reach 5% of 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 that separate them from Sequential Model Based Optimization can make a significant difference.

4.3.5 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 4.2, in terms of distance to the optimum and mean loss.

This figure shows a non negligible 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 Sequential Model Based Optimization. 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.

4.3.6 Impact of the hyperparameters on the heuristics' behavior

The different hyperparameters used to setup the heuristics have an impact on their behavior. Figures 4.3 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

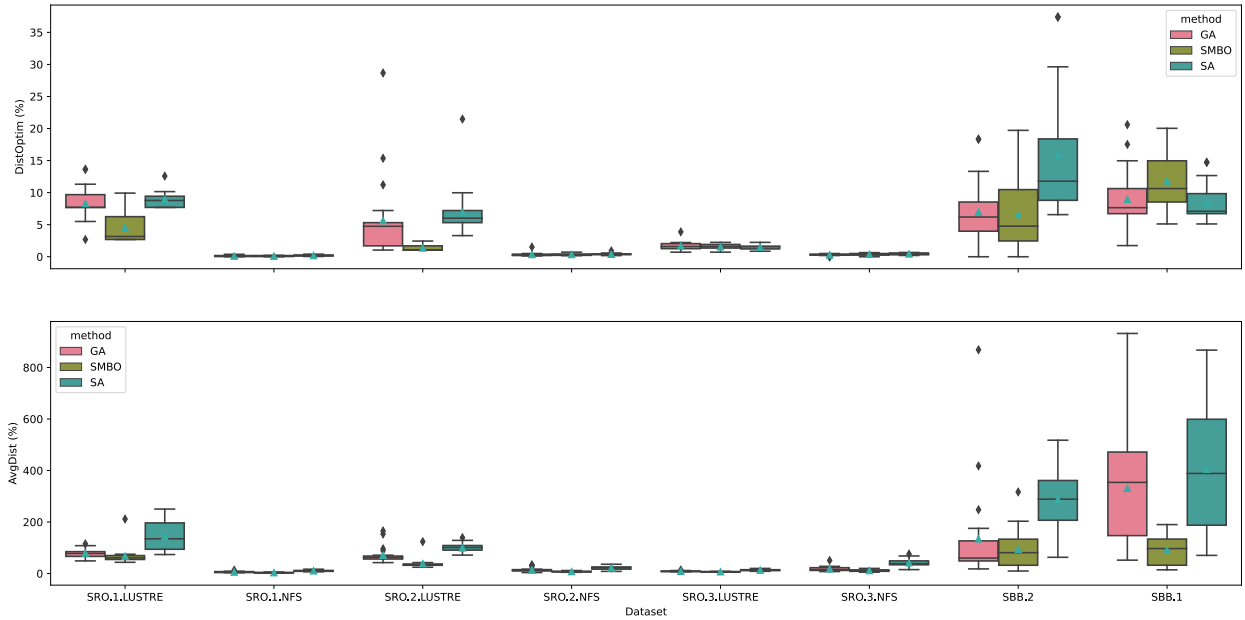


Figure 4.2: Statistical distributions of metrics over the selected heuristics

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.

In the case of Simulated Annealing, shown in figure 4.4, 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.

For Sequential Model Based Optimization, the results in figure 4.5 show that using the Expected Improvement function improves the quality of the optimization, but also reduces its stability. This is consistent with the used algorithm, as expected improvement is considered to be more efficient when looking for the optimum.

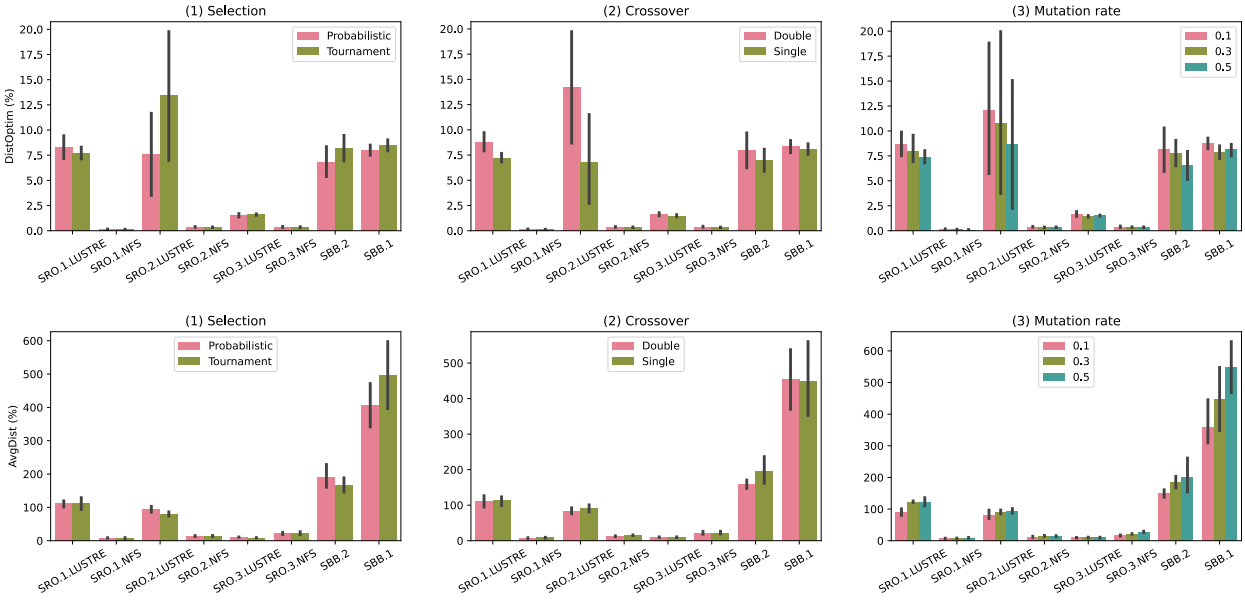


Figure 4.3: Impact of hyperparameters on quality and stability of optimization for Genetic Algorithms

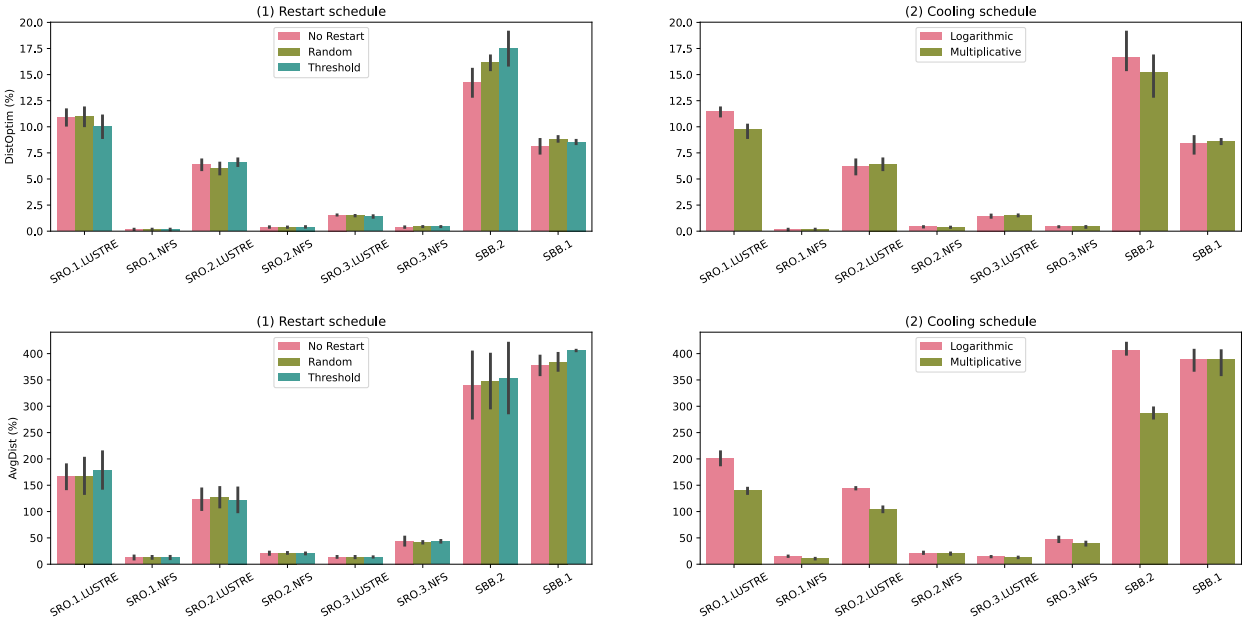


Figure 4.4: Impact of hyperparameters on quality and stability of optimization for Simulated Annealing

4.4 Conclusion

In this chapter, we have successfully demonstrated the usefulness of using black-box optimization for auto-tuning the two I/O accelerators described in chapter 2 when used with I/O intensive applications. We have considered the Sequential Model Based Optimization, genetic algorithms and simulated annealing algorithms, as introduced in chapter 3, with different hyper-parameters in order

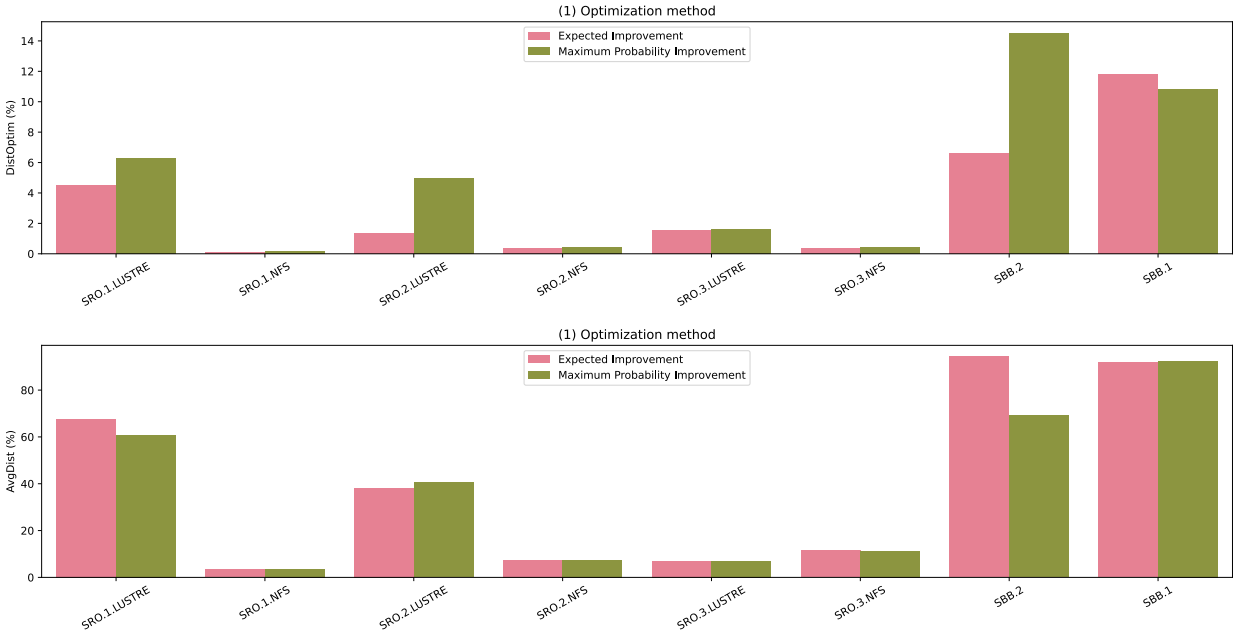


Figure 4.5: Impact of hyperparameters on quality and stability of optimization for Sequential Model Based Optimization

to choose the most suitable one to include in our online tuner. To estimate their suitability, we have ranked them according to the six defined metrics from section 4.7 in chapter 3, which describe their behavior in terms of both optimization quality and stability of the trajectory. This allows to give insight into their relevance for both online and offline tuning, as described in section 1.3 of chapter 1. A comparison to a random sampler has also been added to have a reference ground truth for comparing the heuristics.

For both I/O accelerators, Sequential Model Based Optimization 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 shown that the trade-off between quality and convergence it offers is suitable for the constraints of online and offline tuning. We have also shown its robustness as the two tuned I/O accelerators operate very differently. Because 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.

In this chapter, we operated with the optimistic assumption that the tuning happens on an isolated cluster without any interference, while hpc clusters running in production most of the time have many applications running in parallel, which affects the performance function and transforms the optimization problem into a stochastic one. Indeed, when resources are shared among users, launching the

same job with the same parametrization does not return the same value. Evaluating the impact of noise and adapting the algorithms accordingly is unavoidable for their use in a production setting and this is the main concern of chapter 5.

Bibliography

[1] *Fio benchmark*. https://fio.readthedocs.io/.

[2] *Lustre filesystem*. http://lustre.org/.

[3] J. BERGSTRA, R. BARDENET, Y. BENGIO, AND B. KÉGL, *Algorithms for hyper-parameter optimization*, in Proceedings of the 24th International Conference on Neural Information Processing Systems, NIPS'11, 2011, pp. 2546–2554.

[4] J. BERGSTRA AND Y. BENGIO, *Random search for hyper-parameter optimization*, in Journal of Machine Learning Research, vol. 13, 2012, pp. 281–305.

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

[6] 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, 2018, pp. 893–907.

[7] L. LI AND A. TALWALKAR, *Random search and reproducibility for neural architecture search*, in Proceedings of The 35th Uncertainty in Artificial Intelligence Conference, R. P. Adams and V. Gogate, eds., vol. 115 of Proceedings of Machine Learning Research, 2020, pp. 367–377.

[8] R. SANDBERG, D. GOLDBERG, S. KLEIMAN, D. WALSH, AND B. LYON, *Design and implementation or the sun network filesystem*, 1985.

[9] T. SCHMIED, D. DIDONA, A. DÖRING, T. PARNELL, AND N. IOANNOU, *Towards a general framework for ml-based self-tuning databases*, in Proceedings of the 1st Workshop on Machine Learning and Systems, EuroMLSys '21, 2021, pp. 24–30.

[10] D. WOLPERT AND W. MACREADY, *No free lunch theorems for optimization*, in IEEE Transactions on Evolutionary Computation, vol. 1, 1997, pp. 67–82.

Chapter 5

Tuning in the presence of noise

Contents

In the previous chapter, we have successfully demonstrated the efficiency of black-box optimization for tuning the two I/O accelerators described in chapter 2, and selected Sequential Model Based Optimization (SMBO) as the most promising optimization heuristic. However, this comparative study was done under the assumption that the tuned system is deterministic, *i.e.* a given parametrization will always yield the same execution time. While this hypothesis is valid when working in a controlled and exclusive test environment, it does not always hold for HPC resources shared across many users such as storage systems or for complicated workflows that rely on shared data files accessed in parallel.

However, most of the time, the auto-tuning must happen in a shared noisy setting, because making resources exclusive is expensive, especially storage systems. Automatic tuning methods thus must need to take into account for this possible interference on the tuned application, which degrade the performance of classical auto-tuning heuristics. Indeed, because of the stochasticity of the optimization problem, the algorithm may keep as a good parametrization a bad parametrization that just benefited from good condition, for example because the cluster might have been unusually idle. Reciprocally, it can also reject a good parametrization that was evaluated in difficult conditions, for example during a back-up. This phenomenon confuses the optimization algorithm and greatly slows down the convergence process. As we work with a limited number of possible evaluations, this reduces the efficiency of auto-tuning and can cause sub-optimal performance.

In this chapter, we account for the possible noise present when auto-tuning the I/O accelerators and describe some possible techniques to increase black-box optimization heuristics resilience to noise. We give insight on already existing methods, introduce improvements and prove their performance on two different noisy experiments, and confirm the relevance of taking noise into account when tuning stochastic systems, such as HPC clusters.

5.1 Noisy optimization formulation

In order to take into account the possible interference in the collected data, the formulation of the optimization problem given in section 1.4 needs to be slightly modified. We are indeed looking for the optimum of a performance function that can only be accessed through observations tainted by noise.

Formally, the optimizer must now solve the following problem:

$$min\ \mathbb{E}(F(\theta)),\ \theta \in \Theta$$

$$F(\theta) = f(\theta) + \epsilon(\theta, n)$$

with $f(\theta)$ representing the "true" execution time for parametrization θ, while we only have access to the "observed" or "sampled" execution times $F(\theta)$. The noise, which can depend on the parametrization as well as the optimization step, is a function of θ and n, and is represented by $\epsilon(\theta, n)$. We treat the execution time as a random variable whose realization are the elapsed times for each run. The vector corresponding to the different sampled values at parameter θ will be denoted as $F(\theta)_{1\leq j\leq j_\theta}^{j}$, j_θ being the number of samples for parameter θ. The estimation of f at θ is denoted $\hat{f}(\theta)$.

From this definition, we find that in this noisy framework the optimal parametrization is not the one leading to the quickest run but the one corresponding to the optimal execution time on average. This ensures that the optimum is not a result of chance and actually due to the system's parametrization. Throughout the remainder of this chapter, the term "optimum" will refer to the optimal parametrization on average.

5.2 Tuning of noisy systems in the literature

The literature addressing the problem of noise when tuning real systems is surprisingly sparse, as most of the works detailed in section 1.2 do not study the potential interference on the tuned system and do not mention their tuner resilience to noise either. While this is not critical when working on single user systems, such as local hard drives for personal computers [9], it cannot be ignored when working on highly parallel shared systems [33] [20] [21]. Several studies do acknowledge their system's noise [8], but do not provide any practical solution to make the tuner resilient, other than computing the mean or the median of the found best parametrization repeated several times [28] [15]. To our knowledge, the only notable exception is the Baloo framework [14] developed by Grohmann et al. for tuning distributed database systems, which performs adaptive sampling until the confidence interval around the mean is smaller than a set threshold, or until the maximum number of reevaluation allowed per parametrization is reached. This type of adaptive resampling is explored in section 5.3.2, and its main drawbacks are addressed in section 5.4.

While almost non-existent in the system's tuning community, black-box optimization with noisy fitness is a very proficient field when it comes to theoretical research on synthetic benchmarking function. The available research can be separated into two main categories:

Heuristic specific noise reduction Heuristic specific optimization consists in modifying the heuristic itself in order to make it more resilient to noise. For example, in the case of Sequential Model Based Optimization, as defined in section 3.3.2, and especially Bayesian Optimization, it is possible to modify acquisition functions in order to make them handle noisy observations better. For instance, Gramacy et al. in [13] and Vàsquez et al. in [35] use respectively the mean and a quantile as an estimation of the

performance function through Gaussian Processes when using Bayesian Optimization. Letham et al. propose in [19] a novel way of defining expected improvement, called *Noisy Expected Improvement*, using Quasi Monte Carlo simulation. In [16], Huang et al. suggest using the *Augmented Expected Improvement*, which uses a robust estimation of the best performing parametrization by defining the best solution as the one with the lowest β-quantile, with β a configurable value. A similar quantile based approach is proposed by Picheny et al. in [23] where they suggest using the *Expected Quantile Improvement* to select the next data point to evaluate. In [12], Forrester et al. suggest using a reinterpolation procedure which uses the results of a Gaussian Process regressor on noisy observations into another interpolation model which will be used to compute the Expected Improvement. While these different methods have proven to be efficient when facing different noises on different benchmarking functions [24] [17], they have the major drawbacks of being very specific to the selected optimization heuristic, and cannot be generalized to other heuristics, such as genetic algorithms or simulated annealing. As discussed in chapter 4, there is no single best performing heuristic for every optimization problem, and we wish to deal with the noise using methods that allow switching heuristics depending on the context.

Heuristic agnostic noise reduction Heuristic agnostic optimization consists in performing the tuning with no modification of the selected heuristic. These type of methods only modify the way the data points are acquired and the way the fitness function is fed into the algorithm, as described in figure 5.1.

Definition 20: Heuristic specific noise reduction

Heuristic specific noise reduction consists in modifying the black-box optimization heuristic to improve its resilience to noise.

Definition 21: Heuristic agnostic noise reduction

Heuristic agnostic noise reduction consists in adding mechanisms within the optimization process to improve noise resilience, but without modifying the optimization heuristic.

We have identified from the state-of-the-art two of the most popular and efficient agnostic heuristic noise reduction methods: fitness aggregation and resampling.

(a) **Fitness aggregation**: Fitness regression uses parameter combinations that have already been evaluated to estimate the execution time at each parametrization using regression techniques [36]. Either all the previous history or only neighboring parameters can be used for regression. One of the advantages of using only neighboring parametrizations is that any assumptions made by the regressor on the execution function only has to hold locally. If $\mathcal{N}_x$ is a neighborhood of x

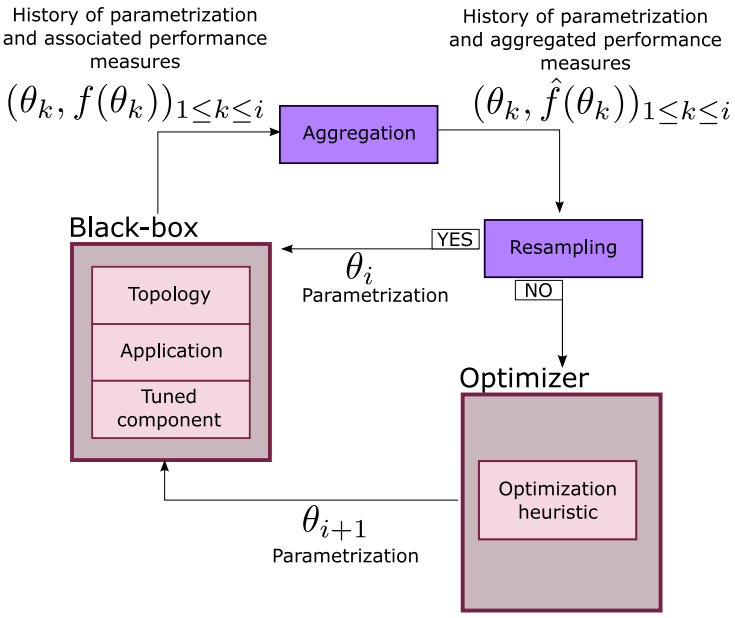


Figure 5.1: Schematic representation of the optimization loop using two noise reduction methods

(that can also be the whole space) and $g_{\mathcal{N}_x}$ a regressor trained on this neighborhood, $f(x)$ can be estimated by:

$$\hat{f}(x) = g_{\mathcal{N}_x}(x)$$

Different regression functions and neighborhood definitions have been suggested in the literature. For example, Kita et al. suggest using *Memory-based Fitness Evaluation Genetic Algorithms* (MFEGA) in [18][27], which consists in approximating the true value of the fitness function as a weighted average of every evaluated parameters. A similar suggestion is made by Branke et al. in [6], where they also perform an estimation of the fitness, but this time using only neighboring data points, which allows to model the noise differently in different zones of the parametric space. In [25], Ratle suggests fitting a Kriging model on the fitness function when using evolutionary algorithms. Finally, in [22], Paenke et al. use a local quadratic regression model to estimate the fitness function.

(b) **Resampling**: Resampling consists in reevaluating several times the same parametrization in order to get a more precise idea of its impact on the real execution time [3] [11] [31]. This method is described in details in section 5.3.

Definition 22: Resampling

Resampling consists in the evaluation of the same parametrization several times to have a more precise idea of the performance for this parametrization in the case of stochastic performance.

While fitness aggregation methods has yielded some good results for improving convergence speed on theoretical problems, as shown by [27] [18], a study of ours on the Smart Burst Buffer (SBB) has shown it to be less efficient than resampling methods [26]. We thus focus our work on resampling methods.

5.3 Resampling methods

Resampling consists in adding a resampling filter by using a set logical rule to select which parametrization to reevaluate. A detailed schematic representation of the integration of resampling within the black-box optimization tuning loop is available in figure 5.2. The general goal of resampling is to reduce the standard deviation of the mean of an objective value in order to augment the knowledge of the impact of the parameter on the performance.

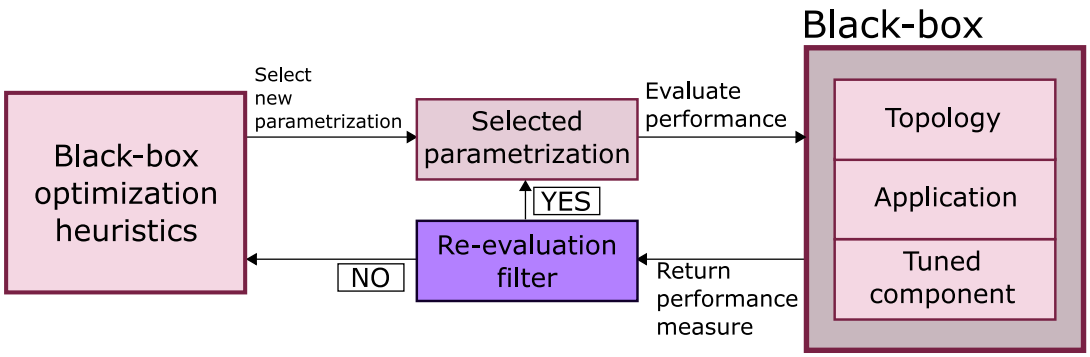


Figure 5.2: Schematic representation of resampling algorithms

Algorithmically, we define a resampling filter as a function $\mathcal{RF}$ which takes as input an optimization's trajectory already evaluated fitness and corresponding parameters $(\theta_i \in \Theta, F(\theta_i) \in \mathbb{R})_k$, for the optimization trajectory at step k, and outputs a boolean on whether or not the last parametrization should be re-evaluated.

$$\mathcal{RF} : (\Theta, \mathbb{R}) \longrightarrow (0, 1)$$

This filter can be integrated for both initialization draws and exploitation draws, or only for exploita-

tion ones. We make the latter choice, as we want to keep the initialization draw to test as many parametrization as possible, and if needed, let the algorithm come back to these parametrization for further investigation. Algorithm 5 describes the integration of a resampling filter with black-box optimization.

Algorithm 5: Black-box optimization with a resampling filter

Data: Target function f; optimization heuristic $\mathcal{H}$; $\mathcal{C}$ a convergence criterion ; possible configuration space; initial design configuration; acquisition function $\mathbb{A}$, resampling filter $\mathcal{RF}$
Result: Optimal configuration

```
i ← 0
S_0 ← IS(Θ)
while C((θ_k, F(θ_k))_{0≤k≤i}) do
    if RF((θ_k, y_k)_{0≤k≤i}) then
        S_i ← S_i   F_i ← F_i ∪ {f(θ_i)}
    else
        θ_i ← H(S_i, F_i)
        S_i ← S_i ∪ {θ_i}
        F_i ← F_i ∪ {f(θ)}
    i ← i + 1
```

Resampling is a trade-off between having a better knowledge of the space and waste some computing times on re-evaluation. Many strategies exist in order to efficiently reevaluate a parametrization, and we will present in this section two of the most popular, simple and efficient resampling algorithms in the literature. For a more exhaustive description of resampling methods, the reader can refer to the thorough classification proposed by Siegmund et al. in [30].

5.3.1 Simple resampling

Simple resampling computes a fixed number of times the fitness value of the selected parametrization [11], regardless of the parametrization and its associated fitness. In our case, it consists in launching a fixed number of times the application and the tuned system with the same selected parametrization. The corresponding resampling filter is algorithmically described in algorithm 6.

Algorithm 6: Simple resampling filter

Data: Number of resamples per parametrization n, Last parametrization θ_j, Already tested parametrization and fitness $(\theta_k, F(\theta_k))_{0\leq k\leq j-1}$
Result: Boolean of whether to re-evaluate or not the parametrization

$|\{\theta \in (\theta_k)_{0\leq k\leq j-1} | \theta = \theta_j\}| \leq n$

The main drawbacks of simple resampling is its lack of adaptivity to the noise present on the parametric space: the need for resampling is most of the time not homogeneously distributed throughout the search space [30]. This is especially true for shared and distributed computer systems, where the noise is often concentrated in a short time span, for example when there is a backup or another

user performing concurrent accesses on the storage systems. To account for this disparity in terms of noise, resampling methods should be able to take into account the performance variance around each data point to adapt the number of evaluations to the system's current state.

5.3.2 Dynamic resampling

Standard Error Dynamic Resampling (SEDR), as introduced by Di Pietro et al. in [10], adapts the number of samples to the noise strength measured at each parametrization. The parametrization is re-evaluated until the 95% confidence interval around the mean is below a fixed threshold τ_{se}. While simple averaging reevaluates each parametrization n times, SEDR introduces a variable number of evaluations n_θ for each parametrization. A schematic representation of the algorithm is available in figure 5.3 and the corresponding algorithm is available in algorithm 7.

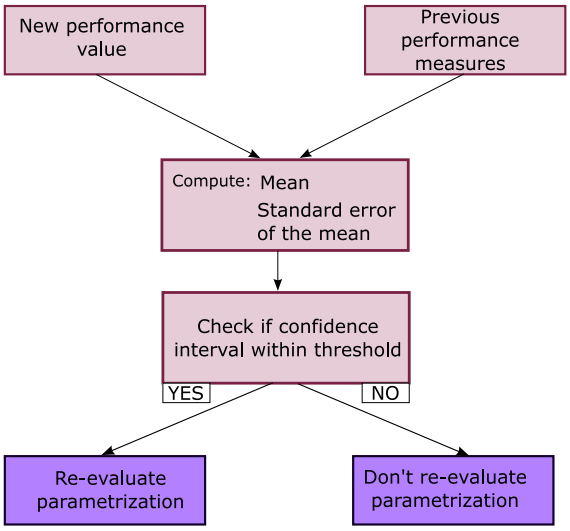


Figure 5.3: Schematic representation of dynamic resampling

We define the function $ci : \Theta \to \Theta$ which computes the 95% confidence interval of the mean for normally distributed data as:

$$ci(\theta) = 2 \times 1.96 \times \frac{\hat{\sigma}(\theta)}{\sqrt{n}} = 2 \times 1.96 \times \frac{\sqrt{\frac{1}{j_\theta - 1} \sum_{j=1}^{j_\theta} (F(\theta)^j - \bar{F}(\theta))^2}}{\sqrt{n}}$$

Algorithm 7: Standard Error Dynamic Resampling filter

Data: Confidence interval threshold τ_{se}, Last parametrization θ_j, Already tested parametrization and fitness $(\theta_k, y_k)_{0 \leq k \leq j-1}$

Result: Boolean of whether to re-evaluate or not the parametrization

1 $ci(\{y \in (_k)_{0 \leq k \leq j-1} | \theta = \theta_j\}) \leq \tau_{se}$

This type of resampling is the one included in the Baloo framework [14], under the name *Adaptive Resampling*, where it has yielded some good results for tuning distributed databases. SEDR has also

proven to be efficient on theoretical problems, as described in the works of Di Pietro et al. [10], as well as practical ones as tested by Syberfeldt et al. in [34].
When using the definition of dynamic resampling introduced by Pietro et al. in [10], the authors do not take into account the fitness value whenever setting a confidence interval threshold and require the same certainty regardless of the performance value. When working with real-systems and especially HPC applications which can take a long time to run, the length of the confidence interval should take into account the application's performance: we do not require the same certainty around the performance of a benchmark running for 10 hours than one lasting for 10 seconds. Because of this, we introduced and used in [26] the definition of an interval width proportional to the currently measured mean for this parametrization, and it is this version of SEDR resampling that will be used during our tests.

5.4 Improving resampling methods

While existing methods have proven to be efficient on several theoretical problems [30] as well as practical ones [4], they present the following drawbacks:

(a) **Resampling can take too many iterations on a single parametrization**: when the noise is very high for a single parametrization, resampling methods based on noise values such as dynamic resampling can spend all their budget on this single parametrization because the deviation is too large. This strong noise can be temporary, for example because the system is going through a particularly high traffic, and the optimizer would benefit more from moving on to another parametrization instead of getting stuck in a locally noisy parametrization. This variation is already included in [14].

(b) **Dependence on hyperparameters**: Dynamic and static resampling have a strong dependence on their hyperparameter (number of resamples in the case of simple resampling and the interval confidence width in the case of dynamic resampling). Indeed, a too high threshold would not trigger any resampling, while a low threshold would trigger too many resampling and prevent the algorithm from exploring the parametric space.

(c) **No comparison with already tested parametrization**: It does not consider how the measured value of the fitness at this data point ranks when compared to the other tested fitness during the resampling process. Because of this, it can resample several times a slow parametrization (compared to the already tested ones) and slows the convergence process of the tuning algorithm. As each run is very costly for HPC systems, this causes waste of resources.

5.4.1 Setting a limit to the number of resamples

To address inconvenient (a), we suggest adding a floor limit N to the allowed number of resamples for a resampling process on a given parametrization, the optimizer being able to come back later to explore more thoroughly this parametrization. As each parametrization is resampled at least twice in the case of dynamic resampling in order to measure the noise on this data point and see if resampling is required, this means that the number of resamples n_θ for parametrization θ is located between 2 and N. This is represented by **component A** in figure 5.5. Throughout the rest of this chapter, we set the maximum number of resamples per parametrization to 10% of the total maximum allowed buget.

5.4.2 Dynamic confidence intervals

Another improvement we suggest is the removal of the dependence on the hyperparameter threshold. To do so, we make the size of the confidence interval dependent on the number of iterations. This addresses the inconvenient (b). The size of the confidence interval becomes inversely proportional to the number of elapsed steps and use a bounded decreasing exponential to select the ratio of the confidence interval at each step, as depicted in figure 5.4. We also make it proportional to the mean measured for this parametrization, to take into account the fact that the required precision on the mean estimator is dependent on the value of the mean.

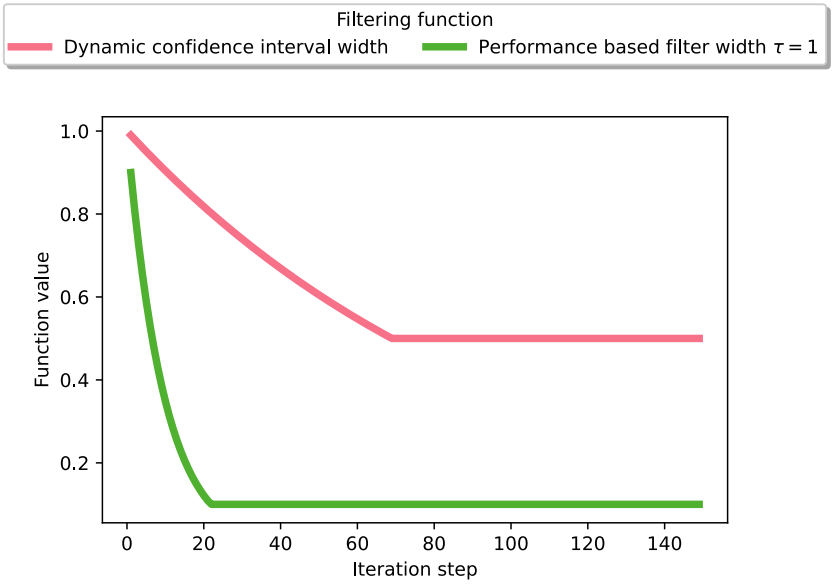


Figure 5.4: Values of filtering coefficients

The confidence interval around the mean thus gets smaller and smaller as the number of iterations raises, as we verify:

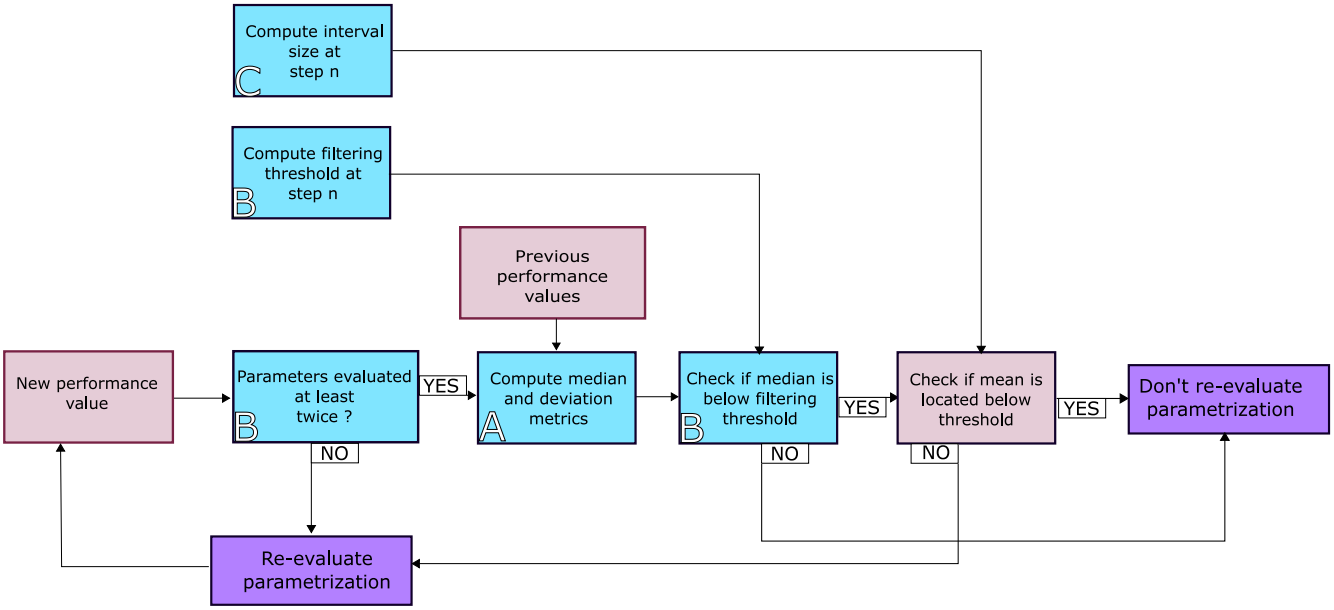


Figure 5.5: Schematic representation of the suggested resampling algorithm

$$ci(\theta) = 2 \times 1.96 \times \frac{\hat{\sigma}(\theta)}{\sqrt{n}} \leq FC(n) \times \hat{\mu}(\theta) = max(0.99^n, 0.1) \times \hat{\mu}(\theta)$$

This ensures that at the beginning of the optimization process, the algorithm is more lax about the precision we want for a parametrization, as we would rather test many different parametrization, ensuring the exploration property of the optimization process, but at the end of the optimization process, we want to be more and more precise about the knowledge we have about the parametrization, ensuring adequate exploitation of already known parametrization. This has three benefits: we keep a strong exploration component of the algorithm at the beginning of the optimization process, we ensure a final knowledge of final parametrizations and we do not have to set any external hyperparameters. We reflect this change in **component C** of figure 5.5.

5.4.3 Performance based resampling filter

The last change we suggest is to introduce a dynamic filtering component before performing the resampling, to address inconvenient (c). Its goal is to filter the most promising parametrizations before submitting them to the resampling which reduces its time cost. This filtering component ensures that we do not waste any optimization budget on slow parametrizations compared to the rest of the evaluated ones. The filtering process runs as follow:

1. Each time the heuristic suggests a new parametrization, it is evaluated at least twice

2. If the median of the performance for this parametrization is inferior to a certain ratio of the cur-

rent median (for example if the ratio is set to 1, we ensure that the new parametrization is better than at least 50% of the already tested parametrizations), move to the next step, otherwise move to step 4

3. Keep this parametrization and submit it to the resampling process.

4. Discard this parametrization as its evaluations are not promising, go back to step 1 if the budget is not empty.

The ratio of the median used in step 2 can either be a fixed value or can be computed dynamically as a decreasing function of the number of elapsed iterations, similarly to dynamic interval definition. Any decreasing function can be used, and we selected a bounded decreasing exponential function as depicted in figure 5.4. As the optimization progresses, this filter ensures that the algorithm is more and more strict about the quality of the resampled solutions, so that last iterations are not wasted on parametrization that are not promising. As the optimization process draws to an end, we make sure that we do not waste any of the remaining resources. This filtering component is reflected by the **components B** in figure 5.5.

More formally, if we denote as FC the function selected to perform the filtering, we ensure that each time a new parametrization θ is selected at step n, it is evaluated at least twice and re-evaluated only if the median performance $med((F(\theta)_i)_{1\leq j\leq n_\theta}^j$ for parametrization θ_i verifies at step n:

$$med((F(\theta)_i)_{1\leq j\leq n_\theta}^j) \leq FC(n) \times med(F(\theta_i)_{1\leq i\leq n}) = max(0.99^n, 0.5) \times med(F(\theta_k)_{1\leq k\leq i-1})$$

where $med(F(\theta_i)_{1\leq i\leq n})$ corresponds to the median of the whole other performance measures up to this new parametrization.

5.5 Experiment plan

To evaluate the relevance of noise reduction and compare our suggested method to the state-of-the-art for the tuning of the two I/O accelerators described in chapter 2, we design two different experiment plans, one for each accelerator. These experiments are designed to perform the optimization in a realistic context with different noise profiles, each representative of conditions encountered when tuning in production.

5.5.1 Tuning with concurrent accesses: the SRO case

The first experiment plan tunes the SRO accelerator with the fakeapp benchmark, in a noisy setting created by performing concurrent accesses on the file read by the benchmark. This type of direct

interference is common when running applications with different concurrent nodes that require using the same data, for example when running complicated workflows. It creates an intense, localized noise, that affects periodically and negatively the system's performance. We describe this experiment plan by first presenting the selected optimized benchmark and its tuning potential when run without any noise, and then explain the noise generation process and its impact on the system's performance.

Tuned application

I/O pattern The selected I/O pattern is a variation of those described in section 2.2.3, using the fakeapp benchmarking application. The application performs operations of size 4k, with 500 hotspots of width 50M, on a file of size 500GB. Within each hotspot, 10000 operations are performed in a random fashion, before moving on to the next file zone. This type of pattern is representative of an application accessing different zone of the data file, for example in the case of a 3D model, and performing many random accesses around this file zone if found interesting, before moving on to the next zone.

Hardware The benchmark runs on a single compute node. It consists in an Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz with 16 physical cores (32 logical cores), bi-socket, and 82 GB of DDR4-DRAM. The back-end parallel filesystem is a Lustre bay [2] of 40TB. The storage system is isolated from the rest of the users in order to have a fine control over the generated noise.

Tuning potential without noise To find the optimal parameters of the accelerator for the benchmark in order to use them as the ground truth, we run some auto-tuning optimization experiments 20 times on isolated nodes and storage system to remove any possible interference. While the resampling methods are heuristic agnostics, we have to select one for the optimization process, and we choose to use SMBO with Expected Improvement and Gaussian Process Regression based on the results found in chapter 4. The initialization plan uses LHS as described in section 3.2, with 10 iterations for initialization. The maximum number of iterations is set to 100 optimization steps. The selected stop criterion, as described in section 3.4, is improvement based, and the optimization process stops if there is less than 5% of improvement in the optimum over 15 iterations. The optimization grid is available in table 5.1.

Table 5.1: Optimization grid for the SRO experiments

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

A representation of the statistical distribution of the results of the optimizer as well as the improve-

ment over the default parametrization are reported in figure 5.6. The main values of the statistical estimators are available in table 5.2.

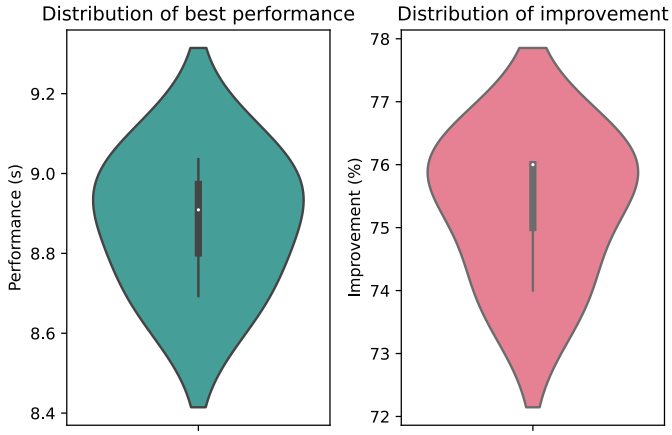


Figure 5.6: Statistical distribution of optimization trajectory descriptors without noise

Table 5.2: Main values of statistical estimators on the optimizer performance without noise

	Average noise (s)	Best fitness (s)	Improvement over default (%)	Convergence speed
Mean	0.0	8.88	75.33	39.67
Standard error	0.0	0.17	1.15	14.57
Minimum	0.0	8.69	74.00	28.00
Maximum	0.0	9.04	76.00	56.00
Median	0.0	8.91	76.00	35.00

We deduce from these results that the optimal execution time is stable across all experiments, with a standard error inferior to 1 second, showing that all optimization converge towards the same optimal performance. The average potential tuning improvement, corresponding to the distance to the default parametrization, is located around 75.33%, with a standard error of 1.15%. The control on our tuning environment is confirmed by these small standard errors measured around the performance values. When it comes to the convergence speed, the behavior differs across the different optimization experiment, with a large standard error of 14.57 steps. However, the maximum number of iterations to reach the optimum is equal to 56 and will be taken as the reference upper bound for the worst case scenario in terms of number of number of steps to reach the optimum value. The retained best performance, used as ground truth for the optimum performance of the application, is taken to be the average value of the best performance over all 15 experiments: 8.88 seconds.

Generated noise

To test the behavior of noise reduction methods in different use-case, we create some noise during the tuning process by performing some concurrent accesses on the same file used by the application during the optimization run.

Generation methodology The noise is created by running in parallel of the main application applications on other computing nodes performing some random operations on the same data file. Three nodes are used, two nodes are performing write operations and one read operations. The parametrization of the fakeapp benchmarks, as defined in section 2.2, are available in table 5.3. These applications correspond to performing random accesses (reads or writes depending on the application) across the whole 500GB file.

Table 5.3: Parametrization of concurrent fakeapp

Access type	Number of operations	Scatter width	Number of leads	Successive operations per lead	Lead advance	Operation size
Write	1000	500GB	0	0	0	$\mathcal{U}(4K, 5M)$
Read	1000	500GB	0	0	0	$\mathcal{U}(4K, 5M)$

These nodes have the same configuration as the one running the main benchmark and described in paragraph 5.5.1. The elapsed time between the arrival of each interference is defined as the time interval between the beginning and the end of the concurrent noise application. It is set to a constant value and is expressed in seconds in table 5.4. Three different noise frequency have been tested. They have been selected to match a certain frequency of arrival relative to the duration of the execution time of the I/O benchmark with default parameters. In order to improve clarity, we will denote each SRO experiment as `SRO.n`, n being the time of arrival of the noise. For example, `SRO.60` will refer to the SRO experiment with interference coming every 60 seconds.

Table 5.4: Frequency of the noise

Elapsed time between arrivals	Frequency relative to the application run
60s	Twice per default optimization run
120s	Every 4 default optimization runs
300s	Every 10 default optimization runs

Noise characterization The impact of the noise on the constant default parametrization is represented in figure 5.7 and the measured noise is available in table 5.5. We can see from this figure that the noise profile is different depending on the frequency of arrival. For a time of arrival of 60 seconds, we see that we create a constant noise, where each run is translated to a higher value. In the case of a time of arrival of 120 seconds, the noise is less constant, but the value of the application does not return to its true value between each run, creating a confusing landscape for the optimizer. In the case of a time of arrival of 300 seconds, we observe a Cauchy-type noise, with a run taking longer than the others when hit by the noise but returning to the original value between each run.

We can see that this noisy environment provides different types of challenge for the optimizer: dealing with a constant noisy environment (60 seconds arrival), an unstable noisy environment (120

Table 5.5: Average measured noise per noise characteristics

Time of arrival (s)	Average noise (s)	Distance to default (%)
60	2.04	93.18
120	5.65	81.18
300	4.56	84.80

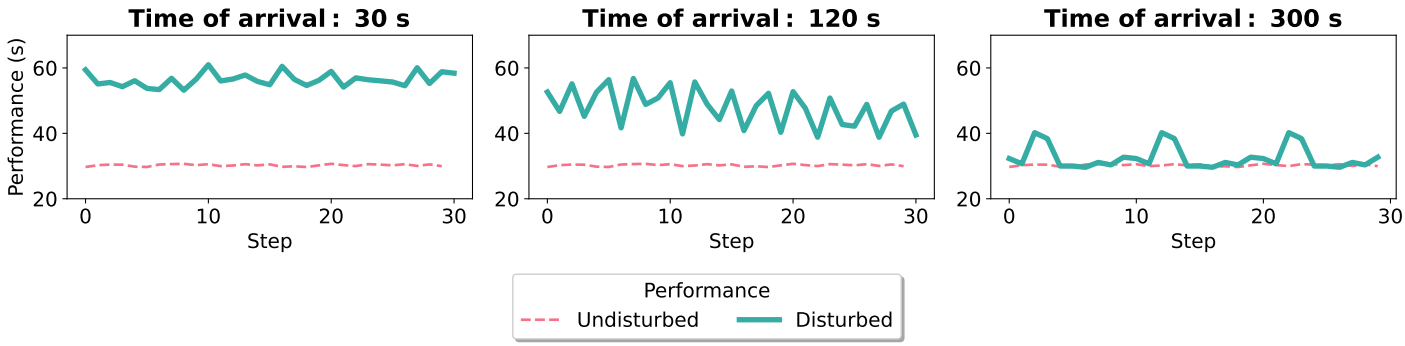


Figure 5.7: Impact of noise on constant parametrization

seconds arrival) and an impulse-type noise (300 seconds arrival).

Tested methods

For the optimization process, we select the same heuristic parametrization as described in paragraph 5.5.1: we choose to use SMBO, using Gaussian Processes as the regression method and Expected Improvement as the acquisition function, the initialization method uses Latin Hypercube Sampling (LHS), the number of initialization runs is set to 10. The maximum number of optimization runs is set to 100, resulting in a maximum total number of iterations of 110. The stop criterion set to stop the experiment automatically when there is less than a 5% improvement over the last 15 iterations. Each optimization process is repeated 5 times to average some of its random behavior. We compare the improvements suggested in section 5.4 to the two most used resampling methods of the state-of-the-art described in section 5.3, as well as when not using any noise reduction strategy. The tested hyperparameters for these methods are available in table 5.6. For our suggested improvements, we compare the impact of adding a dynamic confidence interval and a performance resampling filter to the state-of-the-art. As one of the major advantage of our methods is the absence of hyperparameters, we do not test different hyperparameters values for our solution.

Table 5.6: Tested hyperparameters for state of the art algorithms for the SRO experiment

Method	Hyperarameter name	Tested values
Static resampling	Number of resamples	$\{1, 3, 5\}$
Dynamic resampling	Confidence interval width	$\{10\%, 30\%\}$

5.5.2 Tuning with external noise: the SBB case

In the case of SBB, the noise generation strategy is different and instead of generating it manually, we use the noise naturally present on a cluster shared by twenty users, over a twenty days period. During this time frame, we exhaustively measured the performance corresponding to different parameters repeated several times, in order to obtain noisy measurements, and build a cold dataset with several performance measurements per parametrization.

Tuned application

I/O patterns The application used for evaluation is the standard I/O benchmark IOR, commonly used for evaluating the performance of parallel file systems [1]. We choose to use this benchmarking application instead of the FIO one described in 2.3.3, because of its simplicity of use, as we do not require a fine control over the performed I/O. We perform a write sequential pattern, writing a total of 174 GB using 32 parallel I/O processes, each process writing to its own file by blocks of 1MB. This scenario simulates a write-intensive phase of a standard HPC application's checkpoint creation.

Hardware The IOR benchmark uses three compute nodes and one data node (where the burst buffer's cache is allocated). Each node consists of an Intel(R) Xeon(R) CPU E5-2670 with 16 physical cores (32 logical cores), bi-socket, and 62 GB of DDR4-DRAM. The compute nodes are connected to the data node through FDR RDMA (56Gb/s) network. The back-end parallel filesystem is a Lustre bay [2] of 40TB. The storage system is shared with other users to allow interference and I/O variability.

Tuning potential The default parametrization, which sets the number of workers and destagers to 10, the cache to its maximum size and the cache threshold to 90% yields a mean execution time of 172.69s, which is 12% away from the optimal configuration.

The optimal configuration in terms of cache size is, as expected, the biggest possible (60GB). The default number of workers is also the optimal possible value. However, setting the number of destagers to the maximum possible value, as would be the most intuitive decision based on field-expertise, is actually counter-productive, as the found optimum is equal to 8, and emptying the data node once its 50% full yields the best results.

Data collection

Test parameters To build the exhaustive test dataset, a subset of the most influential parameters described in table 2.3 and validated by the experiments in chapter 4 are tested. This results in a total of 540 tested distinct parametrizations, available in table 5.7.

Each parametrization is sampled 16 times to evaluate the effect of noise on the system. This number was chosen as a trade-off between statistical significance and realistic experimental time

Table 5.7: Tested values for the burst buffer parameters

Parameter name	Range
Number of workers	$\{2, 4, 6, \ldots, 12\}$
Number of destagers	$\{2, 4, 6, \ldots, 12\}$
Cache size	$\{40GB, 50GB, 60GB\}$
Cache threshold	$\{50\%, 60\%, 70\%, 90\%\}$

constraints. According to the notations introduced in section 5.1, this corresponds to collecting m runtimes $F(\theta_i)_{1\leq j\leq m}^{j}$ for each of the n parametrizations $(\theta_i)_{1\leq i\leq n}$ of the burst buffer. In total, the collection of this exhaustive test dataset required 474 hours.

Loop simulation To compute the metrics on realistic trajectories, the auto-tuning loop must be simulated offline to confront the heuristics to real-life conditions. To do so, we use the generated dataset as a black-box, similarly to the methodology used in chapter 4. At each iteration, the heuristic suggests a new parametrization and the dataset is queried to return a randomly selected execution time among those available for this parametrization. The dataset is then used to replace the black-box (*i.e.* the combination between the burst buffer, the application, and the execution topology) in the real-life loop. This approach has the advantage of being very fast to use for tests, because of the cold dataset, as opposed to running the noise reduction optimization experiments directly on the cluster as in section 5.5.1. It has however the drawbacks of being a simplified version of reality, because of the limited 16 sampled performance measure per parametrization, instead of the infinity of possible measures observed in practice.

Description of the SBB dataset

Main characteristics The main statistical estimators and the statistical distribution of the raw collected times and the elapsed times averaged for each parametrization are shown in table 5.8 and a graphical representation of the noisy performance as a function of the tested parameters is available in figure 5.8.

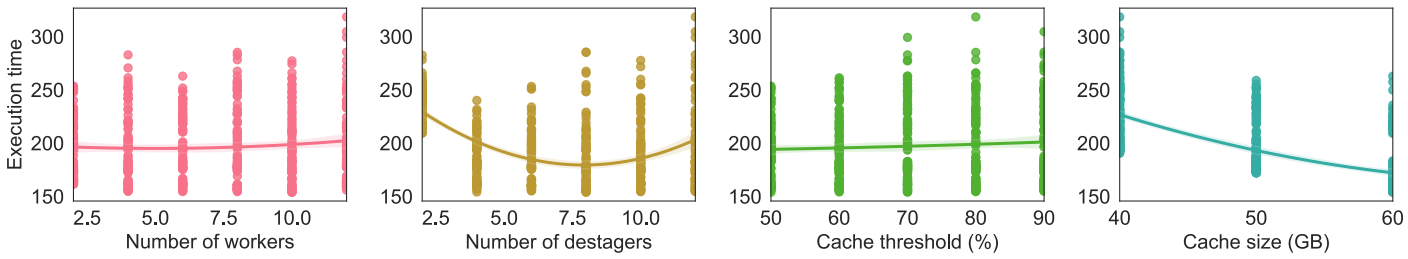


Figure 5.8: Impact of parameters on IOR performance with SBB

Table 5.8: Statistics on raw and averaged elapsed times within each parametrization

	Raw elapsed time	Average elapsed time
Size	8640	540
Mean (s)	197.33	197.33
Standard error (s)	49.49	34.60
Minimum (s)	136.00	151.75
Maximum (s)	625.00	327.37
Median (s)	185.00	188.81

Noise characterization From table 5.8, we deduce that the coefficient of variation is approximately 25% in the case of the raw elapsed times and 17% on the averaged elapsed times. This indicates a strong noise in the dataset to challenge the optimization heuristic. The distribution of this noise is not homogeneous within each parametrization as determined using Levene's and Bartlett's variance homogeneity test ($p - value < 10^{-5}$ in both cases) [5] [7]. These two tests assess the null hypothesis that the variance is homogeneous within each group. This validates the assumption that the noise is not constant over the parametric space and is thus a function of x. An ANOVA [32] which determines if the mean of the noise is constant across each parametrization, shows that some parametrizations are more sensitive to interference than others (p-value < 0.05) .

On average, one outlier run within each parametrization can be detected using Tukey's fences [29], but the distribution of the outliers are even across each parametrization (ANOVA, p-value < 0.05): no parametrization is more susceptible to have outlier runs more than another. These outliers can thus be attributed to the cluster and not to the burst buffer, validating the relevance of the experiment.

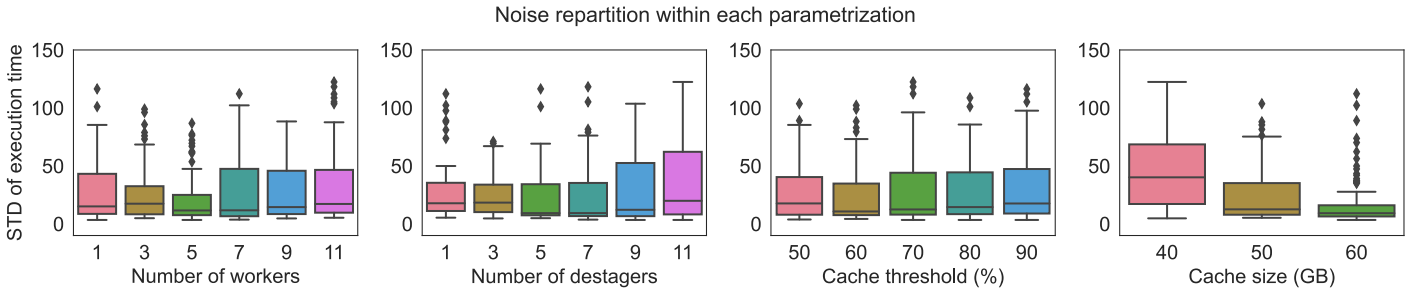


Figure 5.9: Distribution of the standard error within each parametrization

Tested methods

As the cold nature of the dataset makes the optimization runs cheaper to compute, we can compare more hyperparameters of the noise reduction methods. These hyperparameters are available in table 5.9. Similarly as the SRO experiment, we use SMBO, using Gaussian Processes as the regression method and Expected Improvement as the acquisition function. The initialization method uses LHS. The number of initialization runs is set to 10. The maximum number of optimization runs is set

to 150, resulting in a maximum total number of iterations to 160. The optimization process stops if there is less than 5% of improvement in the optimum over 15 iterations. Each optimization process is repeated 30 times to average the possible random behavior.

Table 5.9: Tested hyperparameters for state of the art algorithms for the SBB experiment

Method	Hyperarameter name	Tested values
Static resampling	Number of resamples	$\{2, 5, 10, 15\}$
Dynamic resampling	Confidence interval width	$\{1\%, 5\%, 10\%, 30\%, 50\%\}$

5.5.3 Evaluation metrics

To evaluate the relevance of each optimization heuristic, we compute different metrics described in table 5.10.

Table 5.10: Evaluation criteria for noisy optimization

Name	Abbreviation	Description
Distance to optimum parametrization	AvgDistOpt	Difference between the *average* of the found parametrization to the *best time on average*
Improvement compared to the default parametrization	ImprovementDefault	Difference between the *average* of the found parametrization to the *average performance found for the default parametrization*
Convergence speed	Convergence	Number of elapsed iterations when the stop criterion stops the experiment
Total duration	Duration	Total duration of the experiment as the sum of the execution times at each iteration
Number of triggered resamples	Triggers	Number of times the resampling mechanism is triggered
Number of unique parametrization	Uniques	Number of unique parametrization evaluated

The first metric $AvgDistOpt$ is computed by measuring the distance between the average performance corresponding to the found parametrization and the optimum. This metric corresponds to measuring the asymptotic quality of the optimizer, and is the adaptation to the stochastic case of the $DistOpt$ metric from table 3.1. In the case of the SBB dataset, it consists in computing the mean over the 16 repetitions, while in the case of the SRO, it consists in running 15 times the parametrization found in the noisy case on the noiseless cluster and computing the mean. The metric $ImprovementDefault$ corresponds to the distance between the asymptotic value of the parametrization returned by the tuner and the average performance measured at the default parametrization. It represents the interest of using an auto-tuner rather than simply using the default parametrization. The

third and fourth metrics $Convergence$ and $Duration$ reflect the time taken by the optimization experiment before the automatic stop criterion stops it, both in terms of steps and in terms of elapsed time. There is a trade-off between the convergence speed and the quality of the optimization, as the more we perform evaluations and resampling, the more we gain insight on the system's behavior, but the more expensive resources are spent. We are thus looking for an equilibrium between wanting to have a solution close to the optimum, while minimizing the resources cost.
The two last metrics are specific to the resampling contexts, as they give some insight on the behavior of the resampler. $Triggers$ computes the number of times a resampling cycle is started, corresponding to a number of evaluations above 2 for a given parametrization. The last one, $Uniques$ is closely related to the number of triggers, as it computes the number of unique parametrization evaluated during the optimization process.

5.6 Results and discussion

The results are organized by first presenting the metrics computed on optimization trajectories without any noise reduction, and using state-of-the-art resampling methods. We then present the results obtained when using the improvements we suggested over the state-of-the-art in section 5.4.

5.6.1 On the importance of noise reduction

The values of the different metrics for the two experiments when using Bayesian Optimization without any noise reduction technique are presented in table 5.11.

Table 5.11: Optimization results without noise reduction for the two I/O accelerators

Experiment	Time of arrival (s)	AvgDistOpt (%)	ImprovementDefault (%)	Convergence	Duration (s)
SRO	**60**	75.54	56.70	22.00	1200.98
	120	319.92	-3.58	21.00	1079.57
	300	66.25	58.99	26.00	827.03
SBB		12.90	-0.60	30.73	5728.82

For both experiments, we see that in two cases the noise makes the auto-tuner useless if not using any noise reduction: in the case of the SBB experiment, as well as the `SRO.120` one, where the parametrization returned by the optimization process performs worse than the default parametrization, making the optimization experiment a waste of time and resources. Experiments `SRO.60` and `SRO.300` bring a slight improvement compared to the default parametrization of respectively 56.70% and 58.99%, but nothing as high as the 75% that can be expected, as can be seen from the far distance between the returned optimum and the actual optimum.
These results show that the noise makes the optimization problem a lot more difficult, even in the case

of lower frequency noise, such as `SRO.300`. The main reason for this difficulty is that it obstructs the values taken by the performance function during the optimization process and prevents the regression model from representing correctly the performance landscape, as well as the acquisition function from computing accurate probabilities. Another reason why the noise affects the quality of the optimizer so much is its impact on the stop criterion: as the optimizer fails to find any improvement in the tested parametrization, the stop criterion is reached and the optimization process stops. This behavior can be seen by looking at the convergence speed of the optimization algorithms from table 5.11, which are much shorter than when in an undisturbed environment, especially in the case of the SRO. For all three experiments, the number of steps required for convergence goes from an average of 39.67 to less than 26 for every time of arrival.

This behavior is illustrated in figure 5.10, where an example of optimization with and without noise are displayed. In the case of noisy optimization without noise reduction, the performance loss caused by the noise is interpreted by the stop criterion as a poor optimization potential. As no significant improvement is seen on the optimization trajectory, the optimization process stops and returns a badly optimized parametrization.

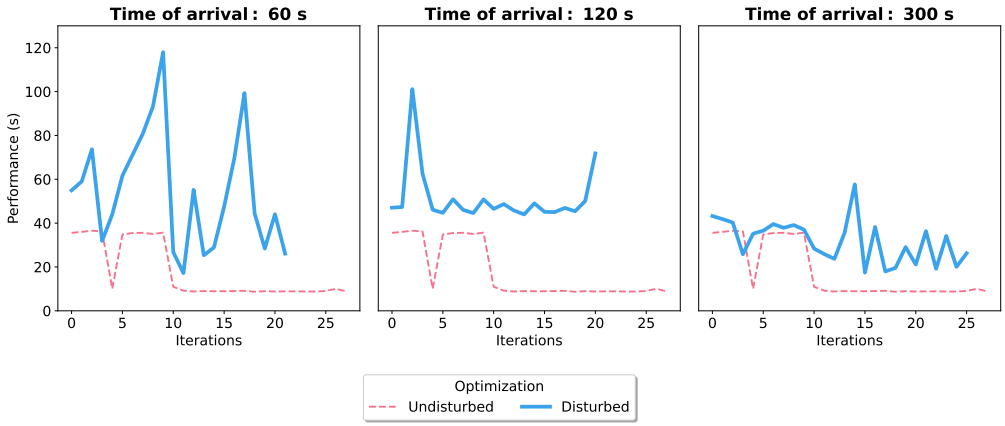


Figure 5.10: Impact of noise on optimization trajectories for the SRO experiment

These results highlight the importance of taking into account the noise whenever performing an optimization in a noisy setting, both in the case of a constant noise as with the SBB experiment and in the case of a regularly arriving noise with the SRO experiment.

5.6.2 Performance of existing methods: static resampling

The metrics computed for the different number of resamples in the case of static resampling are available in table 5.12 for the SBB and in table 5.13 for the SRO.

When comparing the results of static resampling compared to not using any noise reduction, we see a clear improvement for every experiment in terms of the quality of the optimization performed

Table 5.12: Results for static resampling on the SBB dataset

Number of resamples	AvgDistOpt (%)	ImprovementDefault (%)	Convergence	Duration	Uniques	Triggers
2	8.19	3.60	49.13	8924.86	29.43	19.57
5	6.86	4.78	100.33	17948.20	28.03	18.07
10	6.71	4.92	160.00	28204.71	24.90	30.00
15	13.41	-1.05	160.00	29340.22	19.93	30.00

Table 5.13: Results for static resampling on SRO experiment

Number of resamples	Time of arrival (s)	AvgDistOpt (%)	ImprovementDefault (%)	Convergence	Duration (s)	Uniques	Triggers
3	**60**	23.57	69.52	78.01	2430.698	36.01	34.03
	120	30.92	67.71	50.98	1411.23	27.02	20.50
	300	38.65	65.80	51	1211.79	27.03	20.51
5	**60**	15.98	71.39	70	1946.672	22.01	12.0
	120	27.37	68.58	70	1843.915	22.02	12.01
	300	2.84	74.63	90	1632.083	26.12	16.02

by the auto-tuner. Indeed, for every experiment, there exists a number of resamples that improves significantly the performance of the tuner when compared to not using any resampling process, in terms of distance to the ground truth. In the case of `SRO.300`, it even comes close to the optimum and the maximum observed potential improvement compared to default. However, adding static resampling slows down the optimization process, compared to not using any noise reduction, as the stop criterion does not stop the experiment once a resampling process is started.

In the case of the SBB, the best number of resamples is 10, while in the case of the SRO it is 5. This difference highlights the difficulty of using the right hyperparametrization, especially as the noise level depends on the state of the system: performing too many resamples is a waste of resources in the case a system that is not very noisy, while setting a too low number of resamples may waste every optimization attempt in a very noisy system. This highlights the importance of the exploration and exploitation trade-off: too many resamples will give a better knowledge of the performance value, but will prevent the search of a good parametrization as well. For example, in the case of 15 resamples for the SBB experiment, only 20 unique parametrization are evaluated on average, for a performance worse than using the default parametrization. Figures 5.11 and 5.12 present examples of optimization trajectories. The top figure presents the unaggregated optimization trajectories, the middle figure the mean aggregated per consecutive parametrization, for each experiment and their hyperparametrization. They show as well the maximum and minimum performance observed per parametrization, which gives insight of the noise observed at the different steps of the optimization experiment. A bar graph at the bottom indicates the number of resamples per parametrization in the case of the

aggregated trajectories.

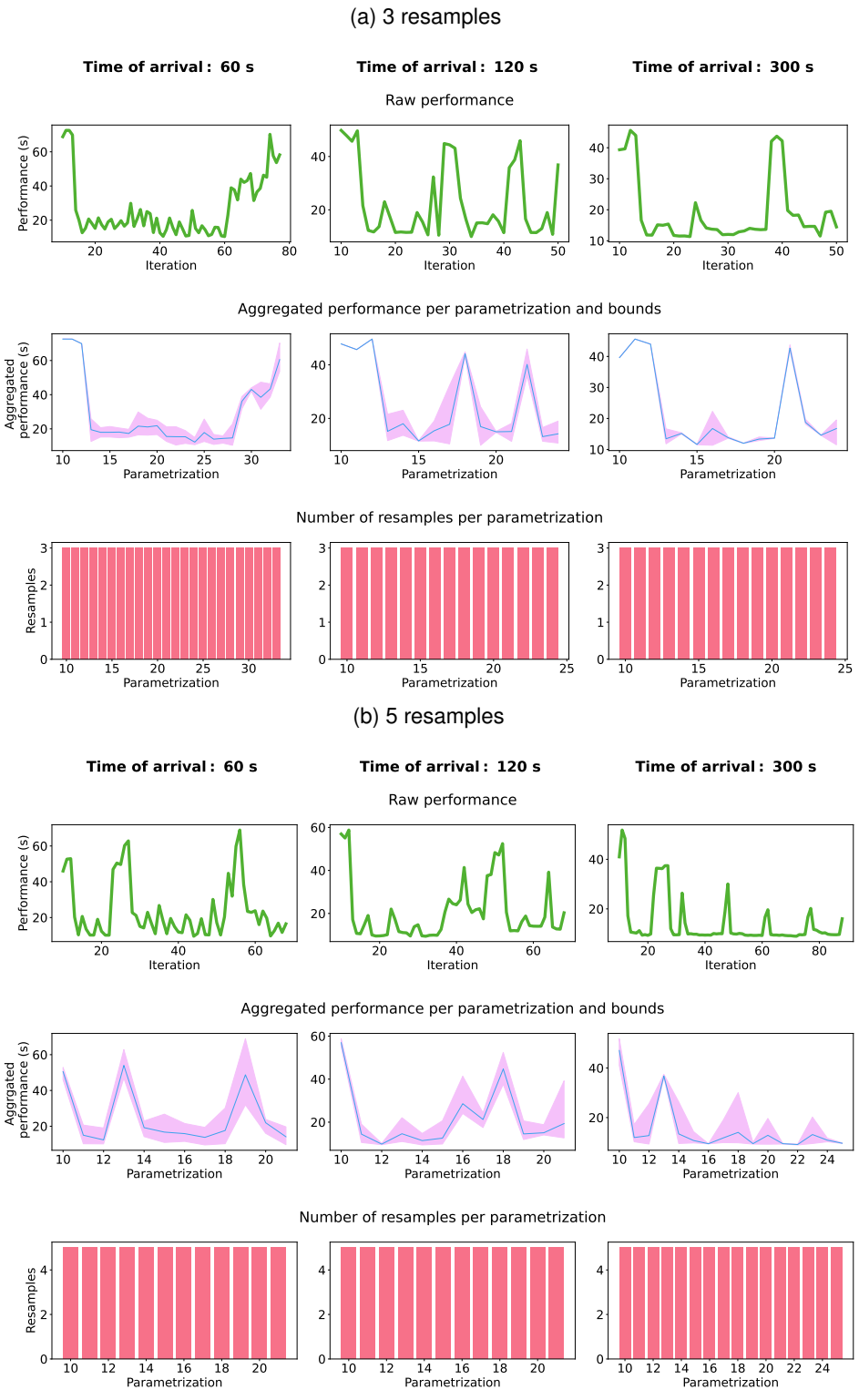


Figure 5.11: Example of trajectories for static resampling on the SRO experiment

These figures highlight one of the most limiting factor of static resampling: it does not take into account the performance bounds whenever selecting the number of resamples. We can indeed see on

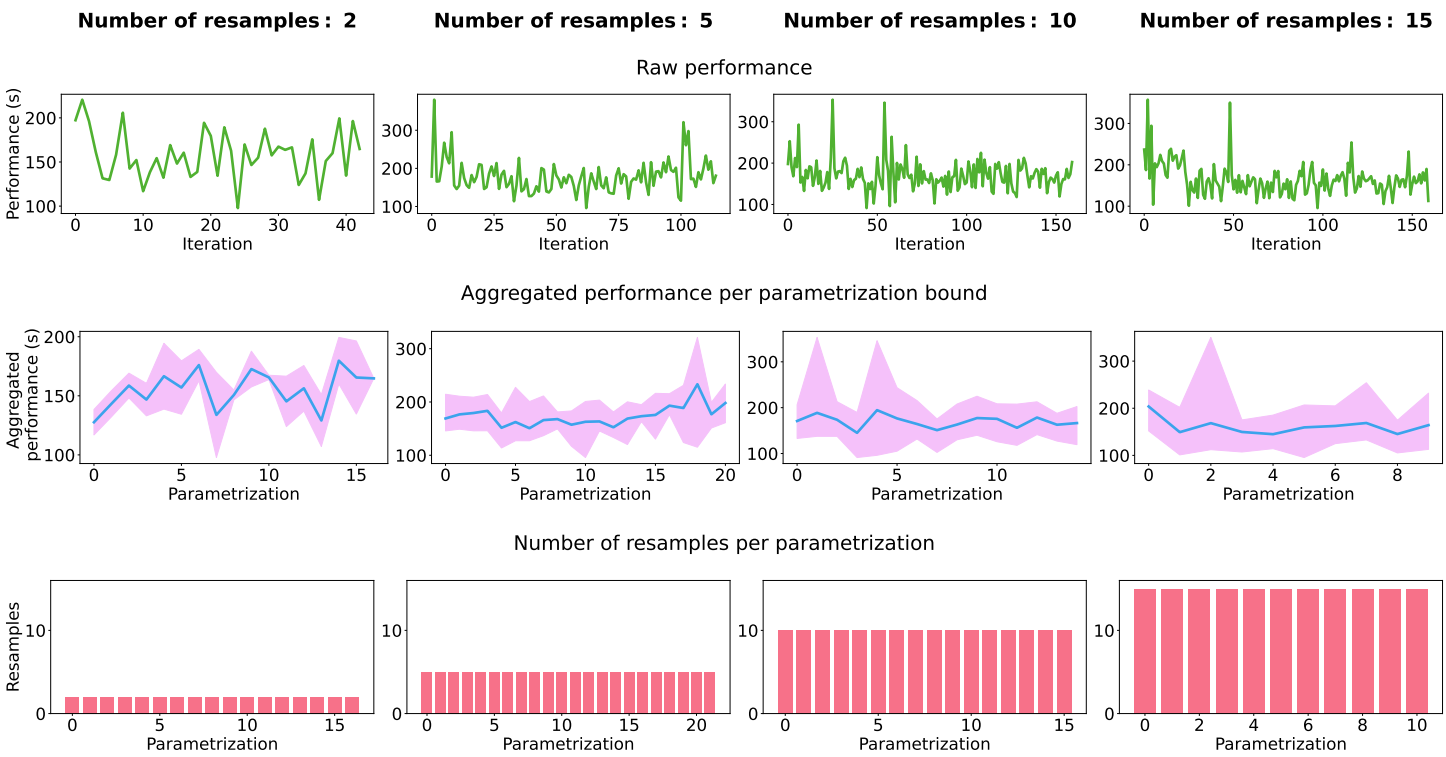


Figure 5.12: Example of optimization trajectory for the SBB using static resampling

these figures that some parametrization have a small bound, because there has been no perturbation during the resampling process, and are re-evaluated the same number of times as some parametrization with larger noise due to on-going interference: static resampling does not take into account the bounds around the performance, as it resamples the same number of times parametrization with a large variance as those with a smaller one. These experiments highlight the main drawbacks of using static resampling, confirming the findings in [30] and the drawback (b) described in section 5.4: static resampling is hard to parametrize in order to find a good exploration-exploitation trade-off and some iterations are wasted on parametrization that do not present a strong noise, while some would have benefited from a more thorough exploration.

5.6.3 Performance of existing methods: dynamic resampling

The results computed by using dynamic resampling based on the standard error of the mean on the SBB experiment are displayed in table 5.14 and in table 5.15 for the SRO experiment.

In the case of the SBB, we see an improvement of using dynamic resampling rather than static resampling, especially in terms of convergence speed: for a performance similar in terms of distance to the optimum, using dynamic resampling with 30% interval width shotens the optimization experiment from 160 steps to 82 steps. This reduces the optimization duration from 28204 seconds (ap-

Table 5.14: Results for dynamic resampling on the SBB experiment

Percentage (%)	AvgDistOpt (%)	ImprovementDefault (%)	Convergence	Duration (s)	Uniques	Triggers
1	12.50	-0.25	160.00	31363.43	10.93	37.07
3	14.42	-1.95	160.00	29243.58	10.97	36.97
5	9.81	2.16	160.00	29485.59	11.00	37.07
10	11.57	0.58	160.00	30601.36	14.20	35.83
30	6.64	4.98	82.03	15657.69	28.30	9.53
50	8.86	3.00	51.20	9349.04	28.03	1.30

Table 5.15: Results of dynamic resampling for the SRO experiment

Percentage (%)	Time of arrival (s)	AvgDistOpt (%)	ImprovementDefault (%)	Convergence	Duration (s)	Uniques	Triggers
10%	**60**	53.99	62.02	100	4304.41	13.05	21.00
	120	112.40	47.61	100	3131.82	15.01	20.05
	300	28.21	68.37	100	1620.60	22.05	17.02
30%	**60**	309.31	-0.96	50	2240.62	30.03	0.00
	120	308.15	-0.68	32	1994.496	21.32	0.00
	300	205.76	24.58	58	2014.72	29.18	2.03

proximately 7 hours and 45 minutes) to 15657 seconds (approximately 4 hours and 30 minutes), with a slight improvement in terms of performance. The interest of using dynamic resampling instead of static resampling for the SBB experiment can be seen in figures 5.14, as we see that parametrization with a small bounds are not resampled while those with a larger one are resampled several times, for example on the figure showing a 10% width resampling interval. We also see from this figure the importance of using the right hyperparametrization of the algorithm: when the filter is too strict (in the case of the SBB below 5%), only a few parametrization are explored, while when the filter is too strict (in the case of the SBB above 50%), parametrization are almost never resampled more than twice and the performance of the optimizer is reduced.

In the case of the SRO experiment, we find no benefit of using dynamic resampling compared to static resampling, and in the case of the 30% interval width, the tuner behaves even worse in terms of optimization quality than when not using any noise reduction strategy. The reasons behind this lack of performance can be observed from figure 5.15, which shows the difficulty of finding the right hyparametrization of the algorithm depending on the noisy setting. In the case of using a 10% interval width, shown in figure 5.11a, with noise with a higher frequency (`SRO.60`), the parametrization are resampled too much, thus testing only 4 different parametrization and not being able to explore the space, while the contrary happens with the noise with a lower frequency, where the algorithm never resamples until a strong noise arrives and the algorithm spends all the optimization budget on this parametrization and stops performing any space exploration. In the case of an interval width of 30%, shown in figure 5.15b, the resampling interval is too large and the opposite behavior is observed: the

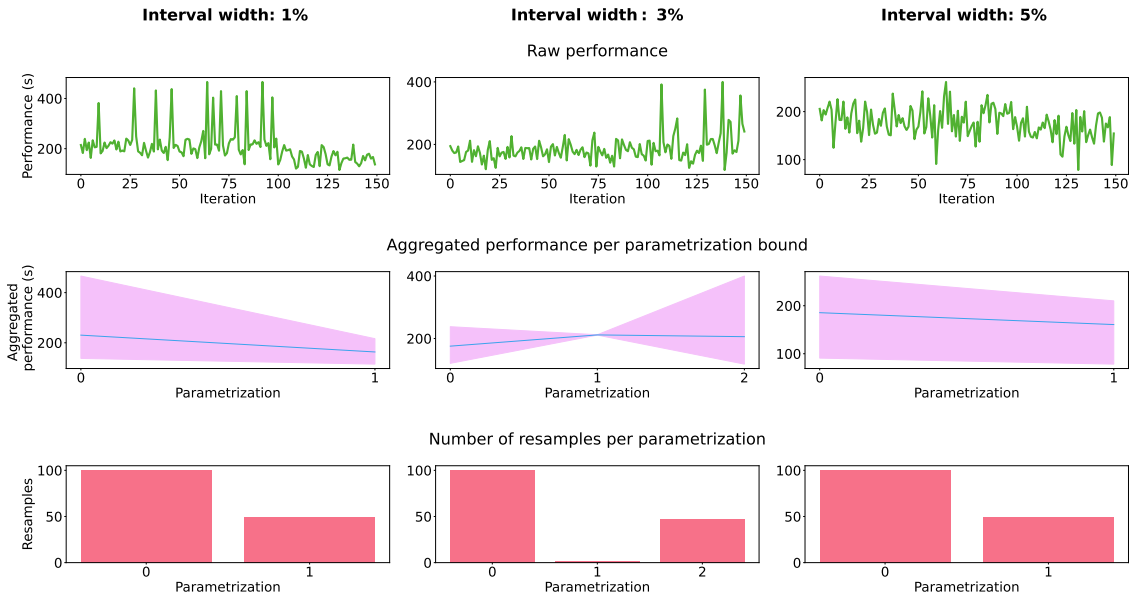


Figure 5.13: Example of optimization trajectory for the SBB experiment using dynamic resampling (interval width set to 1%, 3% and 5%)

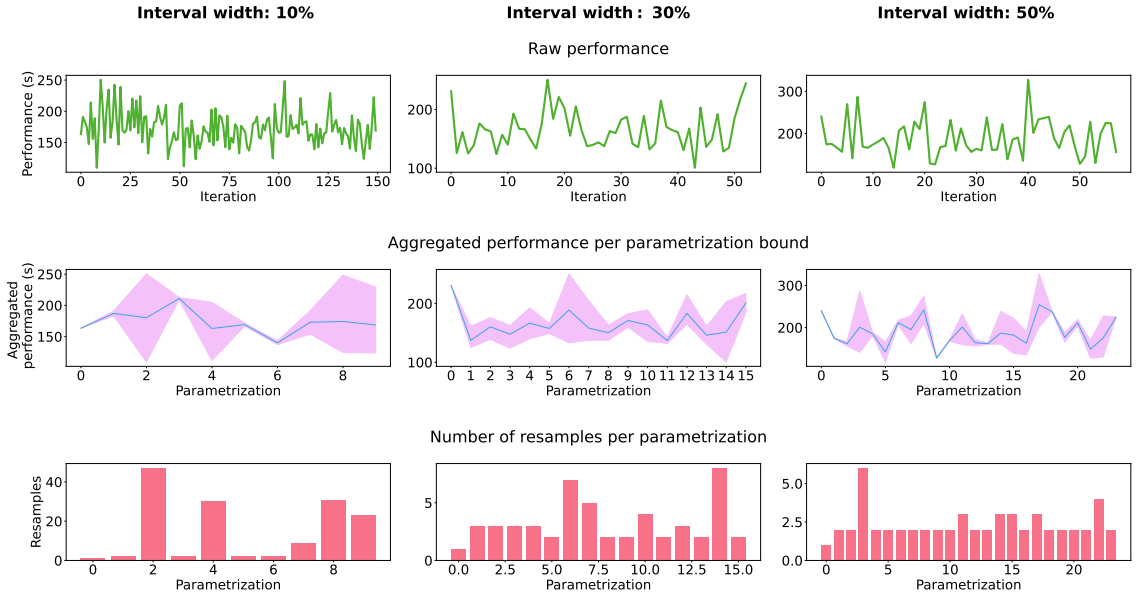


Figure 5.14: Example of optimization trajectory for the SBB experiment using dynamic resampling (interval width set to 10%, 30% and 50%)

parametrization never gets resampled more than twice, which does not give the heuristic enough data to have a good understanding of the performance for this data point. These figures also highlight the advantage of dynamic resampling over static resampling in the case of noise arriving regularly: the resampling process is triggered only periodically whenever the noise affects the optimization process.

In conclusion, over all experiments, we find that dynamic resampling is very sensitive to its parametrization and the relationship between the hyperparameters and the heuristic behavior depends on the

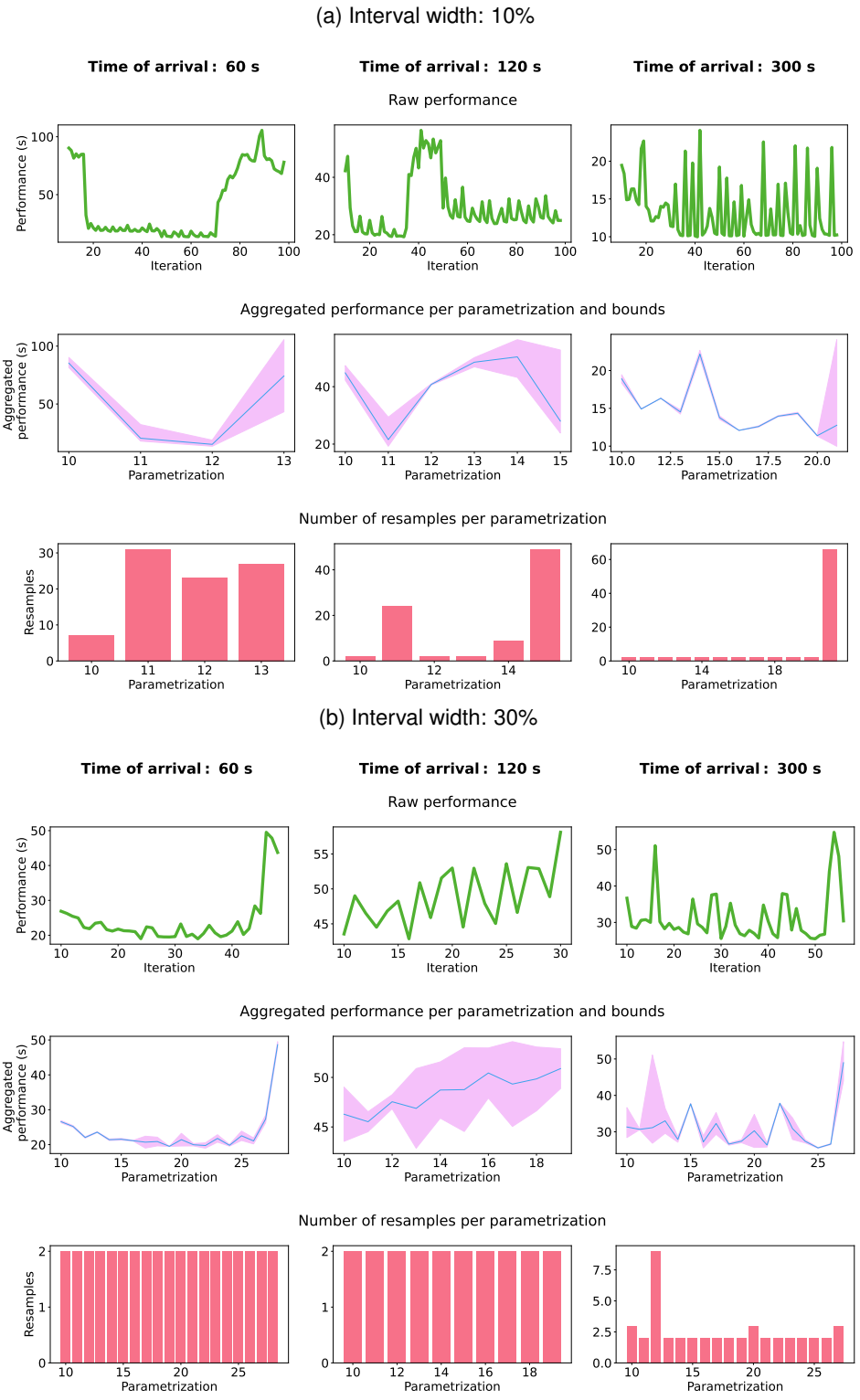


Figure 5.15: Example of trajectories for dynamic resampling on the SRO experiment

tuning problem. This confirms the drawbacks (b) hightlighted in section 5.4. Drawbacks (a) is also observed, as dynamic resampling can spend all of its budget resampling only a few parametrization whenever the noise arrives and neglecting the rest of the space, as highlighted on the SRO experiment

with a 10% interval width.

5.6.4 Using dynamic intervals and setting resampling bounds

The metrics associated with adding the resampling filter, as well as limiting the maximum number of runs, as described in section 5.3.2 are displayed in table 5.16.

Table 5.16: Metrics values for using a dynamic interval definition

Experiment	Time of arrival (s)	AvgDistOpt (%)	ImprovementDefault(%)	Convergence	Duration (s)	Uniques	Triggers
SRO	**60**	8.91	73.14	100.00	2641.19	20.01	16.01
	120	5.12	74.07	100.00	1803.06	23.02	14.05
	300	2.39	74.74	100.00	1517.28	28.45	12.02
SBB		7.38	4.32	160.00	28064.51	23.57	34.63

The first notable advantage of this method is the removal of the need of selecting a particular hyperparameter for the algorithm, which is a difficult task as highlighted by the results of the previous sections. Because the hyperparameter value varies throughout the run, the resampling rate also varies and adapts itself to the requirements of the noisy settings. Bounding the number of resampling iterations is also key to not wasting too many iterations on a single parametrization.

This change in behavior results in a strong improvement of the distance to the optimum compared to using standard dynamic resampling in the case of the SRO algorithm: the distance to the optimum decreases from 53.99% to 8.91% for `SRO.60`, from 112.40% to 5.12% for `SRO.120` and from 28.21% to 2.39% for `SRO.300`. This is reflected as well in the improvement compared to the default parametrization, which is above 73% compared to the default parametrization. For all experiment, in terms of convergence, there is no improvement of using dynamic intervals rather than standard dynamic resampling: the whole budget is spent for every experiment, as the maximum number of iterations is reached for every optimization experiment.

Figure 5.16 provides examples of optimization trajectories: the raw and the aggregated per parametrization execution times are displayed, as well as the number of resamples per parametrization and the width of the interval per optimization iteration. These figures display most importantly the effect of bounding the number of iterations per parametrization, as we see that more different parametrization are tested than in the unbounded case, while still resampling noise-tainted parametrization.

In the case of the SBB experiment, the improvement compared to using unbounded dynamic resampling does not bring improvement compared to using a 30% resampling dynamic interval, as the distance to the optimum increases from 6.64% to 7.38%. However, the suggested method still removes the burdensome task of finding this hyperparameter that was found by running many differently parametrized offline experiment and garantees some adaptivity if the noise were to change on

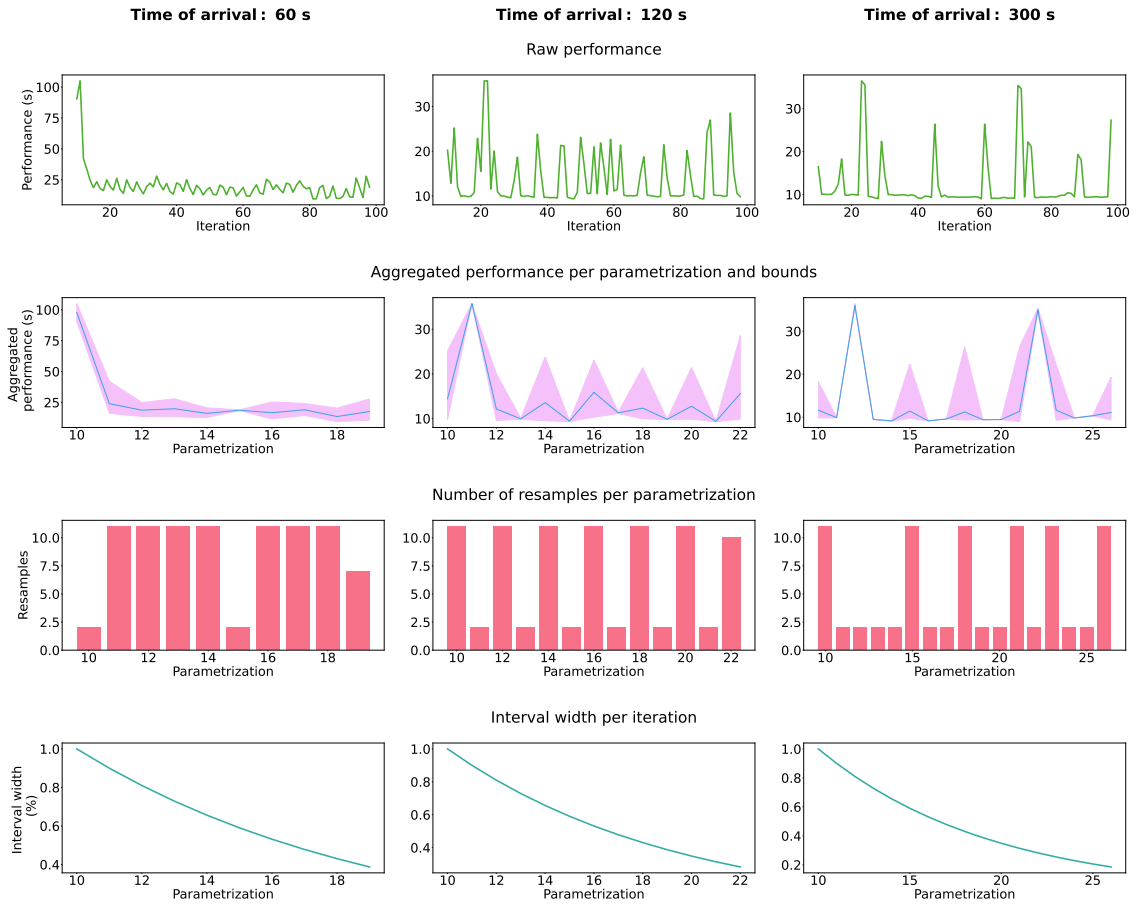


Figure 5.16: Example of optimization trajectory for the SRO experiment using dynamic confidence interval

the system. Similarly to what was observed on the SRO experiment, the optimizer spends the whole optimization budget. An example of optimization trajectory for the SBB experiment, containing the same information than in the case of the SRO experiment, is available in figure 5.17.

We can see from this figure that the resampling process does not start until the space has been explored because of the higher interval width at the beginning of the experiment, but becomes more and more pronounced as the experiment progresses and the interval width gets smaller. The bounds limit the number of iterations per resampling process and thus avoids spending all the budget on a single parametrization, even though the interval width is small.

From these experiments, we deduce that bounded resampling process combines the advantages of static resampling, as no parametrization is resampled too many times, as well as dynamic resampling as we can see from the example trajectories that parametrization without noise are not resampled. Adding a dynamic interval width removes the need of making the difficult choice of selecting the right hyperparametrization and enables the algorithm to find a correct trade-off between exploration of new data and exploitation of already known promising zones.

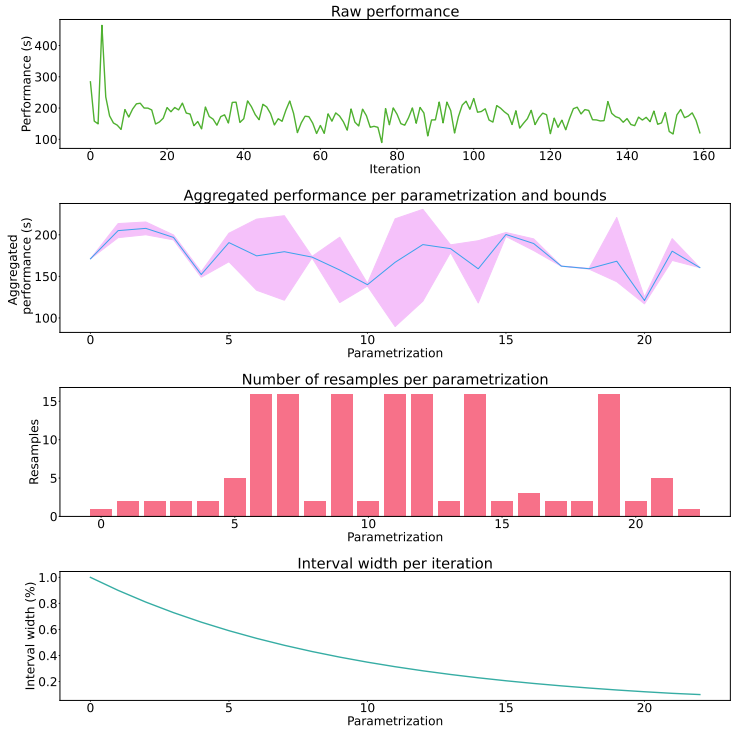


Figure 5.17: Example of optimization trajectory for the SBB experiment using dynamic confidence interval

5.6.5 Adding resampling filters

The values of the metrics computed on the experiments of using all of the improvements suggested in section 5.4 are available in table 5.17.

Table 5.17: Metrics value when using all add-ons

Experiment	Time of arrival (s)	AvgDistOpt (%)	Improvement-Default(%)	Convergence	Duration (s)	Uniques	Triggers
SRO	**60**	5.81	73.90	100	2045.52	30.21	11.01
	120	4.27	74.28	81.12	1556.82	32.31	6.32
	300	2.57	74.70	75.89	1311.26	38.01	2.02
SBB		5.00	5.05	36.07	6565.91	22.10	0.37

In terms of distance to the ground truth, adding a resampling filter allows to find results very close to the ground truth optimum, that bring a strong improvement compared to the default parametrization. Even in the case of the most noisy optimization problem `SRO.60`, the performance measured at the returned parametrization is only 5.81% away from the ground truth, with an absolute difference of 0.5 seconds, and reaching almost the full optimization potential of the auto-tuner. In the case of `SRO.120`, the performance measured at the returned parametrization is 4.27% away from the optimum, corresponding to a 0.36 seconds difference between the ground truth and the found performance. For `SRO.300`, the returned performance is 2.57% away, for an absolute difference of 0.15

seconds. In the case of the SBB experiment, the distance of the found parametrization compared to the optimum is 5%, corresponding to an absolute difference in execution time of 8.63 seconds. Overall experiments, we find that the distance to the optimum is very small and, especially in the case of the SRO experiment, almost negligible to users.

The most notable difference of adding a resampling filter to dynamic bounded intervals is the improvement in convergence speed, as the filter prevents the evaluation of parametrization that are not interesting in terms of performance. Indeed, when not using any filter, the stop criterion did not stop the experiment before the entire budget was spent, as reported in table 5.16, but using a filter makes the experiment faster in the case of `SRO.120`, with a 18.89% speed-up, `SRO.300`, with a 24.11% speed-up, and `SBB`, with a 77.45% speedup, in terms of number of iterations before the run stops. Even when the number of iterations is not reduced as in the case of `SRO.60`, this speed-up is also reflected on the total duration of the experiment which is reduced for every tested experiment. Indeed, we observe for `SRO.60` a time gain of 22%, for `SRO.120` a time gain of 16%, for `SRO.300` a time gain of 14% and for `SBB` a time gain of 76%. This advantage of resampling can also be found when looking at the variable $Triggers$, which is greatly reduced when using a resampling filter compared to not using any, and the variable $Uniques$, which shows that more unique parametrization are tested.

Visually, this behavior can be seen on the optimization trajectories of the SRO experiments in figure 5.18 and for the SBB experiment in figure 5.19, where we see that the resampling process is only triggered for parametrization that are both noisy and interesting to resample, and that this definition becomes stricter over time.

We see from these figures that this filtering process is particularly interesting in the case of very noisy experiments, like `SRO.60` and `SBB`: it prevents the triggering of the resampling process even though the noise is high, and lets the optimizer explores the space, only allowing resampling when it finds a parametrization which value is interesting.

5.6.6 Comparison to the state-of-the-art

For an easier comparison of our results to the state-of-the-art, the values of the most interesting metrics are available in table 5.18.

In terms of distance to the optimum, available in table 5.18a, we see that our solution performs better than the state-of-the-art algorithms, even when selecting those with the best hyperparameters: static resampling is outperformed by 72.6% and 25.5% and dynamic resampling by 93.5 % and 24.7%, for respectively the SRO experiment and the SBB experiment. We thus suggest a method which both improves the convergence property of noise reduction algorithms and removes the need for

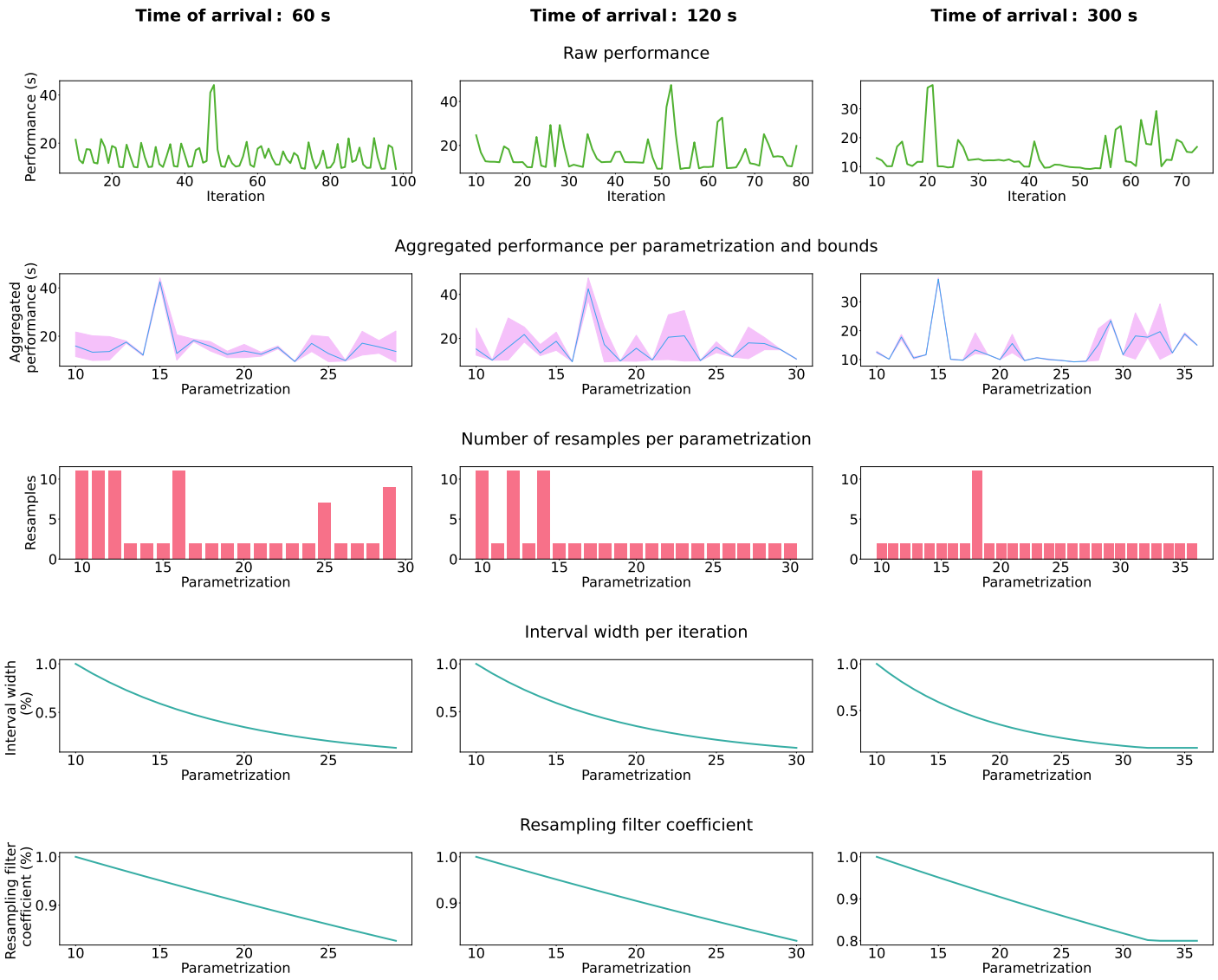


Figure 5.18: Example of optimization trajectory for the SRO experiment using our solution

hyperparameters, primarily by adding bounded dynamic interval resampling to the tuning process. The same improvement is seen when looking at the distance of the turned optimum from the default parametrization in table 5.18b: our solution outperforms existing methods. Another important result of our study is the importance of noise reduction: when not using any, black-box optimization is unable to perform the tuning in a noisy environment, as the returned parametrization brings little to no improvement compared to the default parametrization. Our solution improves the convergence quality of the auto-tuner by 97.46% in the case of the SRO and 61.24% in the case of the SBB rather than when not taking the noise into account.

In terms of convergence speed and experiment duration, the results available in table 5.18c, show a time gain in terms of experiment duration compared to the state-of-the-art, because of the added

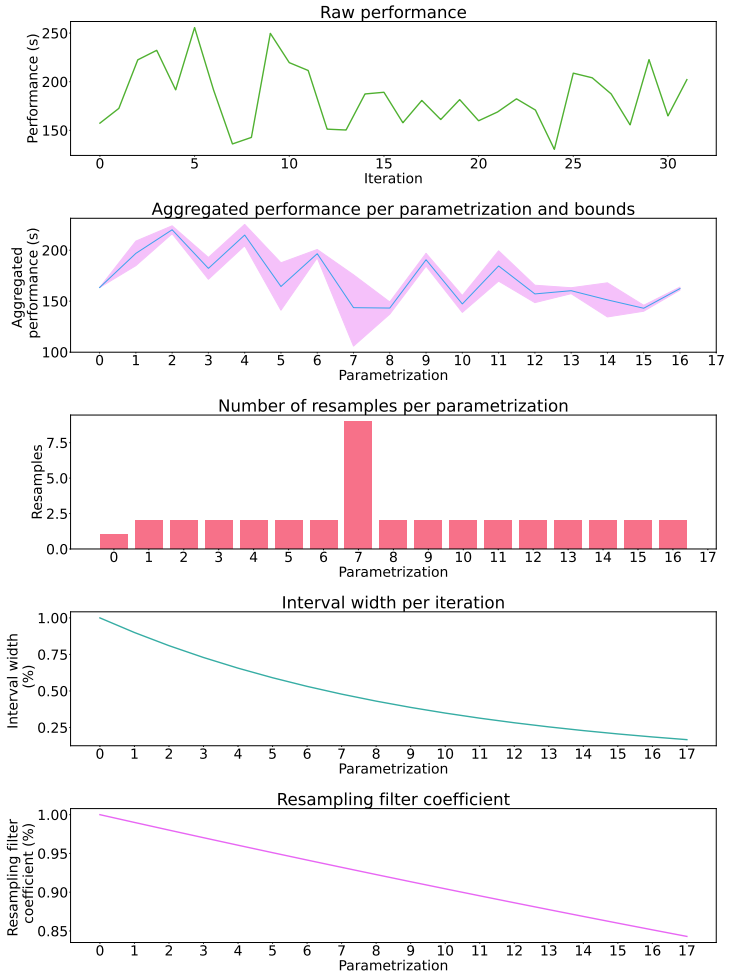


Figure 5.19: Example of optimization trajectory for the SBB experiment using dynamic confidence interval and filtering

resampling decision filter. Indeed, compared to static resampling, we find a time gain of 9.40 % and 76.72 %, and compared to dynamic resampling, a time gain of 45.76 % and 58.07 %, for respectively the SRO and the SBB experiment. Our solution thus improves the convergence speed of the state-of-the-art algorithms.

5.7 Conclusion

This chapter presents a study of the impact of noisy interference on the performance of black-box optimization auto-tuner. By studying two types of different noise and the two different accelerators, we prove that the noise cannot be neglected whenever performing black-box optimization on systems running in production, as it severely impacts the performance of the optimizer.

To increase the resilience of the auto-tuner to the noise observed when running in production, we introduce the concept of resampling and suggest three possible improvements to dynamic resampling, one of the most famous resampling technique. Over four different experiments, we show that

Table 5.18: Comparison of our solution to the state of the art in terms of:

(a) Distance to the ground truth (%)

	SRO	SBB
No noise reduction	153.90	12.90
Static resampling	15.39	6.71
Dynamic resampling	64.86	6.64
Our solution	4.21	5.00

(b) Improvement compared to the default parametrization (%)

	SRO	SBB
No noise reduction	37.37	-0.60
Static resampling	71.53	4.92
Dynamic resampling	59.33	4.98
Our solution	73.88	5.05

(c) Experiment duration

	SRO	SBB
No noise reduction	1035.33	5728.82
Static resampling	1807.55	28204.71
Dynamic resampling	3018.94	15657.69
Our solution	1637.86	6565.91

our solution improves the convergence of state-of-the-art dynamic resampling by 93.5% and 24.7% for respectively the SRO and the SBB accelerator, as well as speeds-up the experiment duration by 45.76% and 58.07% for these same accelerators. We also prove the importance of using noise reduction strategies whenever tuning systems running in production, as we find that using noise reduction strategies increases the found optimum by respectively 97.46% and 61.24%.

Bibliography

[1] *Ior benchmark description*. http://wiki.lustre.org/IOR.

[2] *Lustre filesystem*. http://lustre.org/.

[3] A. N. AIZAWA AND B. W. WAH, *Scheduling of Genetic Algorithms in a Noisy Environment*, in Evolutionary Computation, vol. 2, 1994, pp. 97–122.

[4] H. ANAHIDEH, J. ROSENBERGER, AND V. CHEN, *High-dimensional black-box optimization under uncertainty*, in Computers Operations Research, 2021.

[5] D. H. ARSHAM AND M. LOVRIC, *Bartlett's test*, in International Encyclopedia of Statistical Science, vol. 2, 2011, pp. 20–23.

[6] J. BRANKE, C. SCHMIDT, AND H. SCHMEC, *Efficient fitness estimation in noisy environments*, in Proceedings of Genetic and Evolutionary Computation, 2001, pp. 243–250.

[7] M. B. BROWN AND A. B. FORSYTHE, *Robust tests for the equality of variances*, in Journal of the American Statistical Association, vol. 69, 1974, pp. 364–367.

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

[9] 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, 2018, pp. 893–907.

[10] A. DI PIETRO, L. WHILE, AND L. BARONE, *Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions*, in Proceedings of the 2004 Congress on Evolutionary Computation, vol. 2, 2004, pp. 1254–1261.

[11] J. M. FITZPATRICK AND J. J. GREFENSTETTE, *Genetic algorithms in noisy environments*, in Machine Learning, vol. 3, 1988, pp. 101–120.

[12] E. I. J. FORRESTER, A. J. KEANE, AND N. W. BRESSLOFF, *Design and analysis of noisy computer experiments*, AIAA Journal, pp. 2331–2339.

[13] R. GRAMACY AND H. LEE, *Optimization under unknown constraints*, in Bayesian Statistics, vol. 9, 2010.

[14] J. GROHMANN, D. SEYBOLD, S. EISMANN, M. LEZNIK, S. KOUNEV, AND J. DOMASCHKA, *Baloo: Measuring and modeling the performance configurations of distributed dbms*, in 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2020, pp. 1–8.

[15] Y. GUR, D. YANG, F. STALSCHUS, AND B. REINWALD, *Adaptive multi-model reinforcement learning for online database tuning*, in International Conference on Extending Database Technology, 2021.

[16] D. HUANG, T. ALLEN, W. NOTZ, AND R. A. MILLER, *Sequential kriging optimization using multiple-fidelity evaluations*, in Structural and Multidisciplinary Optimization, vol. 32, 2006, pp. 369–382.

[17] H. JALALI, I. VAN NIEUWENHUYSE, AND V. PICHENY, *Comparison of kriging-based algorithms for simulation optimization with heterogeneous noise*, in European Journal of Operational Research, vol. 261, 2017, pp. 279–301.

[18] H. KITA AND Y. SANO, *Genetic algorithms for optimization of noisy fitness functions and adaptation to changing environments*, in In: 2003 Joint Workshop of Hayashibara 44 Weise, Zapf, Chiong, Nebro Foundation and 2003 Workshop on Statistical Mechanical Approach to Probabilistic Information Processing, 2003.

[19] B. LETHAM, B. KARRER, G. OTTONI, AND E. BAKSHY, *Constrained bayesian optimization with noisy experiments*, in ArXiv, 2017.

[20] O. MONDRAGON, P. BRIDGES, K. FERREIRA, S. LEVY, AND P. WIDENER, *Understanding performance interference in next-generation HPC systems*, 2016.

[21] G. OZER, A. NETTI, D. TAFANI, AND M. SCHULZ, *Characterizing hpc performance variation with monitoring and unsupervised learning*, in High Performance Computing, H. Jagode, H. Anzt, G. Juckeland, and H. Ltaief, eds., 2020, pp. 280–292.

[22] I. PAENKE, J. BRANKE, AND Y. JIN, *Efficient search for robust solutions by means of evolutionary algorithms and fitness approximation*, in IEEE Transactions on Evolutionary Computation, vol. 10, 2006, pp. 405–420.

[23] V. PICHENY, D. GINSBOURGER, Y. RICHET, AND G. CAPLIN, *Quantile-based optimization of noisy computer experiments with tunable precision*, in Technometrics, vol. 55, 2012.

[24] V. PICHENY, T. WAGNER, AND D. GINSBOURGER, *A benchmark of kriging-based infill criteria for noisy optimization*, in Structural and Multidisciplinary Optimization, vol. 48, 2013, pp. 607–626.

[25] A. RATLE, *Accelerating the convergence of evolutionary algorithms by fitness landscape approximation.*, in Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, 1998.

[26] S. ROBERT, S. ZERTAL, AND G. VAUMOURIN, *Using genetic algorithms for noisy systems' auto-tuning: an application to the case of burst buffers*, in Proceedings of the International Conference on High Performance Computing Simulation (HPCS), 2020.

[27] Y. SANO AND H. KITA, *Optimization of noisy fitness functions by means of genetic algorithms using history of search with test of estimation*, in Proceedings of the 2002 Congress on Evolutionary Computation (CEC'02), vol. 1, 2002, pp. 360–365.

[28] T. SCHMIED, D. DIDONA, A. DÖRING, T. PARNELL, AND N. IOANNOU, *Towards a general framework for ML-based self-tuning databases*, 2020.

[29] N. C. SCHWERTMAN AND R. DE SILVA, *Identifying outliers with sequential fences*, in Computational statistics & data analysis, vol. 51, 2007.

[30] F. SIEGMUND, A. NG, AND K. DEB, *A comparative study of dynamic resampling strategies for guided evolutionary multi-objective optimization*, in 2013 IEEE Congress on Evolutionary Computation, 2013, pp. 1826–1835.

[31] P. STAGGE, *Averaging efficiently in the presence of noise*, in Proceedings of the 5th Parallel Problem Solving from Nature, A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, eds., Springer Berlin Heidelberg, 1998, pp. 188–197.

[32] L. STHLE AND S. WOLD, *Analysis of variance (anova)*, in Chemometrics and Intelligent Laboratory Systems, vol. 6, 1989, pp. 259 – 272.

[33] M. STOFFEL, F. BROQUEDIS, F. DESPREZ, AND A. MAZOUZ, *Phase-ta: Periodicity detection and characterization for hpc applications*, in 2020 International Conference on High Performance Computing Simulation (HPCS), 2021.

[34] A. SYBERFELDT, A. NG, R. I. JOHN, AND P. MOORE, *Evolutionary optimisation of noisy multi-objective problems using confidence-based dynamic resampling*, in European Journal of Operational Research, vol. 204, 2010, pp. 533–544.

[35] E. VÁZQUEZ, J. VILLEMONTEIX, M. SIDORKIEWICZ, AND E. WALTER, *Global optimization based on noisy evaluations: An empirical study of two statistical approaches*, in Journal of Physics Conference Series, vol. 135, 2008.

[36] YAOCHU JIN AND J. BRANKE, *Evolutionary optimization in uncertain environments-a survey*, in IEEE Transactions on Evolutionary Computation, vol. 9, 2005, pp. 303–317.

Chapter 6

SHAMan: a flexible framework for auto-tuning HPC systems

Contents

To test the efficiency of the methods described in chapter 3, as well as in chapter 5, a tuning environment must be easily deployed in a production context. For this purpose, we have developed a tuning framework to perform our tuning experiments. As we intend to make the most of the agnosticism of black-box optimization methods, we developed a software as generalist as possible. As

a result, we have bundled the methodology described in the previous chapters into an open-source optimization framework, called Smart HPC Application MANager (SHAMan), which provides an out-of-the-box Web application to perform black-box auto-tuning of custom computer components running on a distributed system, for an application submitted by the user. It has been developed entirely by us and has been made available to the Open-Source community [6] [24].
The framework integrates the three state-of-the-art heuristics described in chapter 3, as well as the noise reduction strategies described in chapter 5 to deal with the possible interference of shared resources for large scale HPC systems, and pruning strategies to limit the time spent by the optimization process. It is to our knowledge the only optimization framework specifically tailored to find the optimum parameters of configurable HPC systems.

6.1 Available auto-tuning frameworks

Several auto-tuning frameworks have been proposed recently in different domain where optimization is required.
The Machine Learning community has proposed several frameworks to find the parameters that return the best prediction scores for a given model and dataset. Among the most popular frameworks, we can cite Optuna [7], which relies on Tree Parzen Estimators to perform the optimization. Autotune [21] is an other framework which supports several black-box optimization techniques. Scikit-Optimize [5] which supports a wide range of surrogate modeling techniques. Hyperopt [10] also supports Tree Parzen Estimator and some variations of this algorithm, with the particularity of supporting asynchronous and distributed optimization as well. The SHERPA [19] library provides different optimization algorithms (Bayesian Optimization, TPE, Random Search ...) with the possibility to add new algorithms. It also comes with a back-end database and a small Web Interface for experiment visualization. Another library available in the state of the art is GPyOpt [9], especially targeted for users who wish to use Bayesian Optimization to plan their laboratory experiments. Orion [11] provides a wide range of optimization algorithms, and has a strong emphasis on the parallelization of optimization runs. Finally, the framework TPOT [22] relies on genetic algorithms for the optimization of Machine Learning pipelines. The main drawbacks identified with these already existing frameworks concern the difficulty of integrating these libraries for purpose other than the ones they were designed for by using them for HPC tuning. It is also difficult to enhance them with other optimization techniques, such as noise reduction strategies. In addition, none of them offer a satisfying Web Interface allowing an easy manipulation of the software.
Another domain which provides auto-tuning frameworks is code and program tuning. Belonging to

this category, we can cite the framework Open Tuner [8] for program auto-tuning. However, these software are very specific to this use-case, and cannot be easily modified for our purpose.
In the MPI community, the two most famous commercical implementations come with their own tuning tool: OPTO [12] is the standard tool used by the Open MPI community for tuning MCA parameters, and similarly mpitune [3] from Intel MPI. These methods only include exhaustive grid search, making these tools slow to use for tuning expensive HPC applications.
Specific to the Hadoop stack, we can mention the tool Starfish [18], which tunes the Hadoop ecosystem according to users' need.
The DataBase Management System community has also developed some tuning frameworks. Notably, we can mention OtterTune [25] [26], which uses Bayesian Optimization and Reinforcement Learning to perform auto-tuning of any configurable database system. A similar approach is proposed in the Baloo framework [17], who models database performance using supervised models. It also adds a noise reduction component to deal with the noisy measurement of distributed system. The tuning framework iTuned [15] is portable across many database systems and uses surrogate modeling to find their optimal configuration.
Frameworks like *Google Vizier* presented in [16] provide a generalist black-box optimization as a service, but it has not be made freely available to the public. Another generalist framework for system's tuning is the BOAT framework [14], but this type of framework requires some insight on the optimized system, as well as some strong coding skills.

6.2 Contributions to existing works

Faced with the highlighted deficiencies of the existing solutions, we have developed our own framework, and provide these main contributions and features:

(a) **Accessibility**: the optimization engine is accessible through a Web Interface

(b) **Optimization diversity**: as different heuristics work differently for different systems, our framework provides several state-of-the-art heuristics

(c) **Easy to extend**: the optimization engine uses a plug-in architecture and the development of the heuristic is thus the only development cost

(d) **Integration within the HPC ecosystem**: the framework relies on the Slurm workload [20] manager to run HPC applications. The microservice architecture enables it to have no concurrent interactions with the cluster and the application itself. It is also not intrusive. As experiments can

be launched by the users on their own, there is no requirement of privileged rights to use the software.

(e) **Flexible for a wide range of use-cases**: new components can be registered through a generalist configuration file.

(f) **Customizable target measure**: the optimized target function can be defined on a case-per-case basis to allow the optimization of various metrics

(g) **Integrates noise reduction strategies**: because of their highly dynamic nature and the complexity of applications and software stacks, HPC systems are subject to many interference when running in production, which results in a different performance measure for each run even with the same system's parametrization. Noise reduction strategies are included in the framework to perform well even in the case of strong interference.

(h) **Integrates pruning strategies**: runs with unsuited parametrization are aborted, to speed-up the convergence process

6.3 Description of the framework

SHAMan is a framework to perform auto-tuning of configurable component running on HPC distributed systems. It performs the auto-tuning loop by parametrizing the component, submitting the job through the Slurm workload manager, and getting the corresponding execution time. Using the combination of the history (parametrization and execution time), the framework then uses black-box optimization to select the next most appropriate parametrization, up until the number of allocated runs is over or the stop criterion is reached.

6.3.1 Terminology

Throughout this section, we will use the following terms:

- **Component**: the component which optimum parameters must be found. It must be configurable, either through environment variables or command line arguments.
- **Target value**: the measurement that needs to be optimized. This target value can be on a component per component basis.
- **Parametric grid**: the possible parametrization tested by SHAMan, defined as a minimum, maximum and a step value.

- **Application**: a program that can be run on the clusters' nodes through Slurm and for which we want to find the optimal parametrization of the component.

- **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 parametrized black-box optimization algorithm that will output the best parametrization for the application and the component.

Definition 23: SHAMan experiment

A SHAMan experiment is defined as the combination of:

- a configurable component
- a target value to measure the performance of the component
- an application to run the component with
- a black-box optimizer algorithm (heuristic, noise reduction strategy, pruning strategy, stop criterion . . .)

The output of a SHAMan experiment are the optimal parametrization, as well as information related to the optimization process, such as the experiments metadata and the performance landscape.

6.3.2 Available methods

The framework comes out of the box with several optimization heuristics, several noise reduction strategies and pruning strategies.

Optimization heuristics The three optimization heuristics (genetic algorithms, surrogate models and simulated annealing) with configurable parameters, as described in depth in chapter 3, are available in the framework. The framework provides abstractions to add new optimization heuristics with minimal development cost.

Noise reduction strategies Three of the noise strategies among those discussed in chapter 5 are available in the framework:

- **Static resampling**: re-evaluates each parametrization for a fixed number of iterations

- **Standard Error Dynamic Resampling**: re-evaluates the current parametrization until the standard error of the mean falls below a set threshold

- **Our improved noise reduction algorithm**: a complex decision algorithm for re-evaluating the current parametrization, described in section 5.4.

Pruning strategies Pruning strategies consist in stopping some runs early because their parametrization is unpromising, compared to already tested parameters. Two pruning strategies are available out of the box in the framework:

- *Default based*: It consists in stopping every run that takes longer than the execution time corresponding to the default parametrization.

- *Estimator based*: It consists in stopping every run that takes longer than the value of an estimator computed on previous runs. For example, if the selected estimator is the median, the current run is stopped if its elapsed time takes longer than 50% of the already tested parametrization. This pruning only applies to runs performed after the initialization ones.

Definition 24: Pruning strategies

Pruning strategies consist in stopping running applications if the parametrization does not seem promising compared to other tested parametrization.

6.3.3 Main features

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

- Define a new configurable component and register it for later optimization

- Design and launch an experiment (as defined above) through a Web interface or through a command line interface;

- Visualize data and results of 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.

Declaring a new component

Running the command `shaman-install` with a YAML file describing a component registers it to the application and makes it possible to optimize this component. This YAML file must describe how the component is launched and declares its different parameters and how they must be used to parametrize the component. After the installation process, the components are available in the launch menu, as seen in figure 6.1. A mock example of an installation file is available in listings 6.1, and the different sections of the YAML file are described in the coming sections. Examples of concrete installation YAML file are available in section 6.5.

Definition 25: SHAMan component configuration file

A YAML file describing how the component is launched and parametrized. This YAML file must be installed before using the SHAMan framework.

Activating the component The possible ways of activating a component is through:

- An option passed on the job's command line (`plugin` section). The job is called using the command `sbatch --plugin example_1`
- a command called on top of the sbatch script (`command` section)
- the setting of a LD_PRELOAD environment variable (`LD_PRELOAD` section) at the top of the script to tune

Commands called between optimization runs The `header` variable is a command written at the top of the batch script generated by SHAMan that will be called between each optimization round, before running the job. For instance, a clear cache command is called when tuning I/O accelerators to ensure independence between runs.

Configuring the component The YAML file also describes how the component is configured and how the values of its parameters are passed to the system. Each parameter must have a default value for SHAMan to compute the performance measure corresponding to the default parametrization and acting as a reference value to compute the improvement brought by the optimization.

The name of the parameters of the component and their types is also described in the YAML file. They can either be:

- Environment variables (`env_var=True`)
- Variables appended to the command line variable with a flag (`cmd_var=True`)
- A variable passed on the job's command line (`cli_var=True`). The job is called using the command `sbatch --param_name param_value sbatch_name`.

Parsing the output The section `custom_target` points to the function to use to parse the output of the script and to output the optimized target. When unspecified, the optimized target is the execution time, as collected by calling the Linux `time` command. The parsing function should be a Python function, located in the `plugins` section of the SHAMan code.

Listing 6.1: Configuration file for install a new component for SHAMan

```
components:
  component_1:
    plugin: example_1
    header: example_header
    command: example_cmd
    ld_preload: example_lib
    custom_target: test_parse_output.test_parse_output
    parameters:
      param_1:
        env_var: True
        type: int
        default: 1
      param_2:
        cmd_var: True
        type: int
        default: 100
        flag: test
      param_3:
        cli_var: True
        type: int
        default: 1

  component_2:
  ...
```

Launching an experiment

Two different ways are available to launch an experiment. The main way is to launch the experiment through the Web interface, but SHAMan can also be launched through a command line interface. We detail the usage of both these features in the following paragraphs.

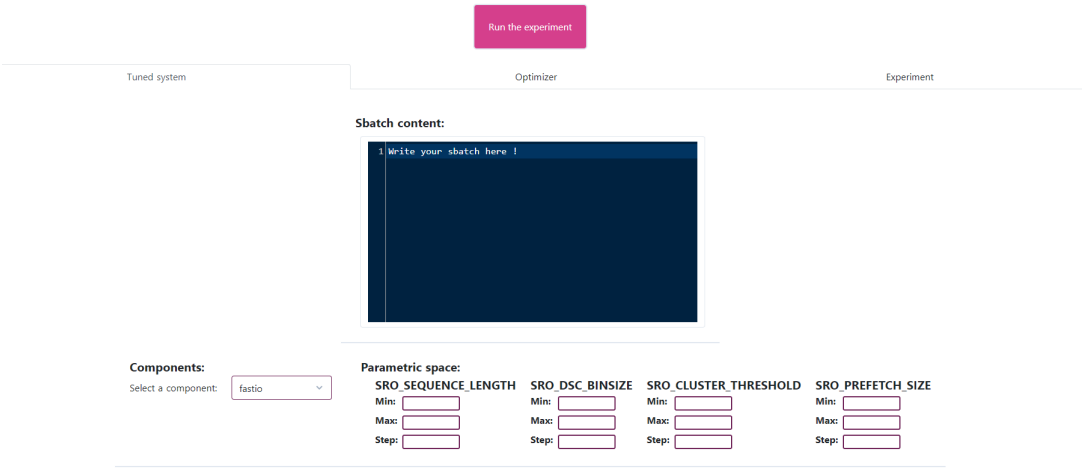


Figure 6.1: Selection of the component and definition of the parametric grid

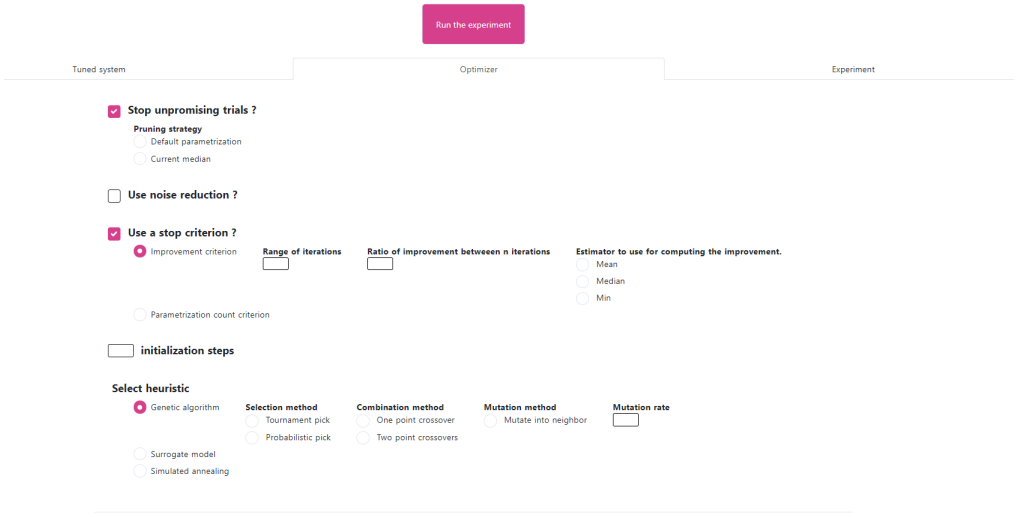


Figure 6.2: Parametrization of the black-box heuristic and the different algorithms

Through the Web interface To launch an experiment through the menu depicted in figures 6.1, 6.2 and 6.3, the user has to configure the black-box by:

1. Write an application according to Slurm sbatch format.
2. Select the component and the parametric grid through the radio buttons (minimal, maximal and step value).
3. Configure the optimization heuristic, chosen freely among available ones. Resampling parametrization, stop criterion and pruning strategies can also be activated.
4. Select maximum number of iterations and the name of the experiment.

Once the appropriate options are selected, the optimization process will begin to run and its information will be available in the exploring section of the Web application.

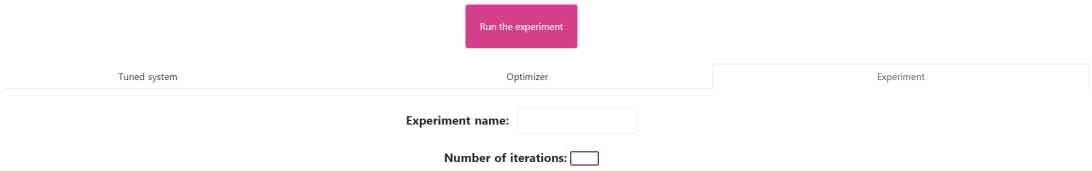


Figure 6.3: Selection of the name of the experiment and the number of iterations

Through a command line interface Another way to use SHAMan is to use it directly through a command line interface, as shown in listing 6.2. It allows more flexibility of the different features but also requires more understanding of the framework's architecture.

Listing 6.2: Using SHAMan through the command line

```
Usage: shaman-optimize [OPTIONS]

  Run an optimization experiment.

Options:
  --component-name TEXT
  The name of the component to tune  [required]
  --nbr-iteration INTEGER
  The maximal number of iterations to run the
                            experiment for.  [required]

  --sbatch-file TEXT
  The path to the sbatch file  [required]
  --experiment-name TEXT
  The name to give to the experiment  [required]
  --configuration-file TEXT
  The path to the configuration file  [required]
  --sbatch-dir TEXT
  The directory to store the sbatch
  --slurm-dir TEXT
  The directory to write the slurm outputs
  --result-file TEXT
  The path to the result file.
```

As detailed in listing 6.2, it requires the same information as the Web interface, with some additional possibilities for logging purpose:

- *The application to optimize*: a file that can be submitted with the *sbatch* command.
- *The number of iterations*: the allocated number of iterations for the experiment.
- *Experiment name*: the name of the experiment for re-identification in the Web interface
- *Configuration file*: the configuration file of the experiment
- *Directory to store the sbatch*: an optional argument to indicate where the sbatch generated by SHAMan must be stored.
- *Directory to store the slurm outputs*: an optional argument to indicate where to store the slurm outputs.

The configuration file of the experiment is a YAML file which allows the user to fine tune the configuration of the optimizer and the experiment. Four sections are available to fill out and control the behavior of the experiment:

(a) `experiment`: contains information about the experiment. Possible options:

- `default_first`: set to True if the first parameter tested by the optimizer

(b) `bbo`: parametrizes the optimizer. The possible options are the different arguments taken by the optimizer, such as its hyperparameters or the convergence criteria.

(c) `noise_reduction`: parametrizes the noise reduction features of the optimizer. The possible arguments are the different noise reduction strategies available in `bbo`, as described in section 5.

(d) `pruning`: parametrizes the pruning strategy features of the optimizer. It is activated by setting the parameter `max_step_duration`, which corresponds to the maximum elapsed time before stopping the parametrization. Its value can be:

- A numpy estimator (for example, `numpy.median`, `numpy.mean`, *etc*). Runs that go above the value of the estimator computed on the already tested execution times are interrupted.
- A float value (for example, 5), which corresponds to an elapsed time of 5 seconds
- The `default` string, which corresponds to stopping runs that take longer than the default value. The option `default_first` in the ‘experiment‘ section must be set to ‘True‘.

(e) `components`: the selected component and the defined parametric grid, using the `min`, `max` and `step` format. If the value is set to ‘multiplicative‘ the step option is used as a multiplicative factor.

Definition 26: SHAMan experiment configuration file

A SHAMan experiment configuration file is a YAML file which defines the parametrization of the black-box heuristic, as well as the parametric optimization grid (as a minimum, a maximum, and step, with the possibility to define it as a multiplicative step).

An example of configuration file is provided in listings 6.3.

Listing 6.3: Example of SHAMan experiment configuration file

```yaml
experiment:
  default_first: True
bbo:
  heuristic: genetic_algorithm
  initial_sample_size: 2
  selection_method: bbo.heuristics.genetic_algorithm.selections.tournament_pick
  crossover_method: bbo.heuristics.genetic_algorithm.crossover.
      single_point_crossover
  mutation_method: bbo.heuristics.genetic_algorithm.mutations.
      mutate_chromosome_to_neighbor
  pool_size: 5
  mutation_rate: 0.4
  elitism: False
noise_reduction:
  resampling_policy: simple_resampling
  nbr_resamples: 3
  fitness_aggregation: simple_fitness_aggregation
  estimator: numpy.median
pruning:
  max_step_duration: numpy.median
components:
  component_1:
    param_1:
      min: 2
      max: 16
      step: 2
```

```
    step_type: multiplicative
  param_2:
    min: 1
    max: 15
    step: 3
```

This particular configuration file available in listings 6.3 launches an experiment which:

- Runs the default parametrization first
- Uses genetic algorithms (tournament pick, single point crossover, random walk for mutation)
- Uses simple resampling, with 3 resampling per parametrization, and the median as an estimator for parametrization with the same parametrization.
- Uses pruning and stops every run that takes longer than the median measured on the data points.
- Uses `component_1` and `param_1` can take any value from 2 to 16 by power of 2, while `param_2` can take any value from 1 to 15 but with an interval of 3 between each value.

Visualization of an experiment

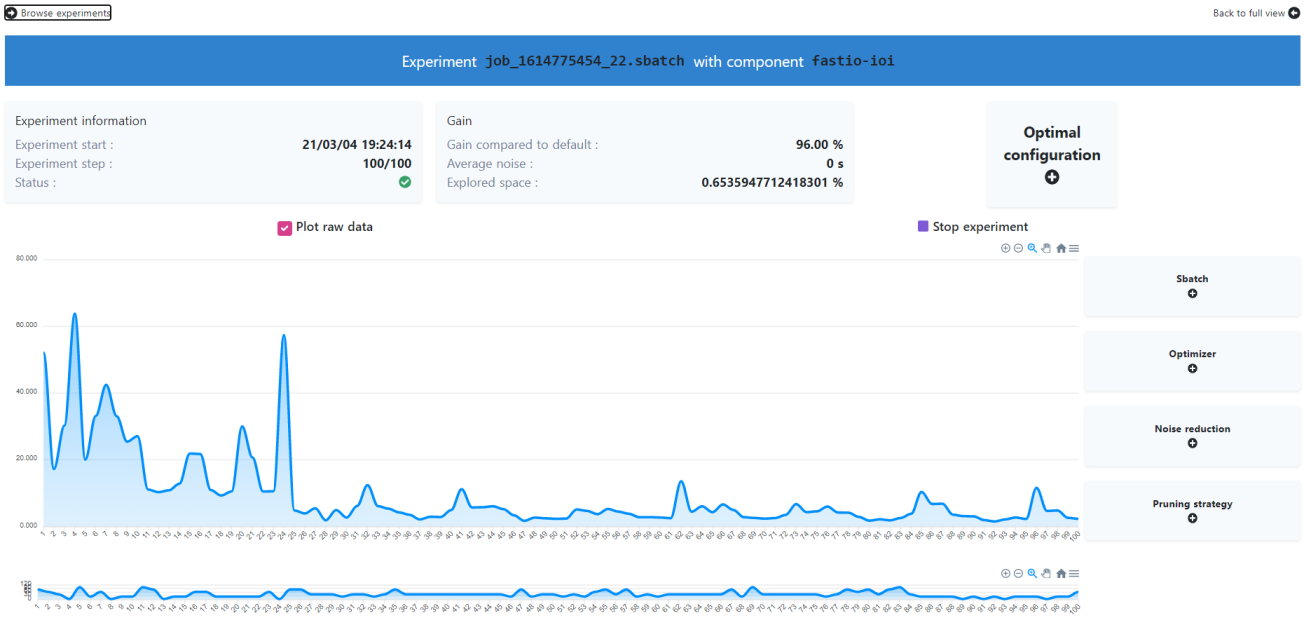
After the submission, the evolution of the running experiments can be visualized in real-time. The optimization trajectory is available through a display of the different tested parameters and the corresponding execution time, as well as the improvement brought by the best parametrization. The other metadata of the experiment are also available through side menus. Examples of visualization are available in figure 6.4a and6.4b. In figure 6.4a, the tuned performance is displayed without any aggregation, but if noise reduction is enabled, the visualization can display the aggregated performance as shown in figure 6.4b.

6.4 Architecture

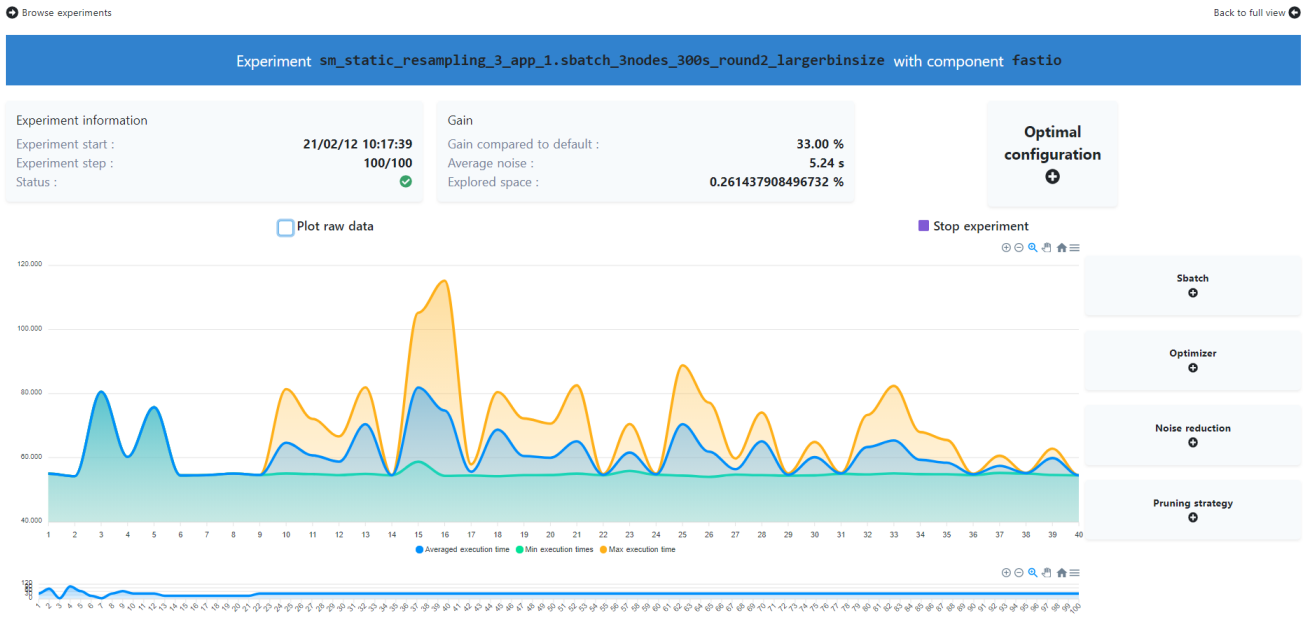
6.4.1 General architecture

The architecture of SHAMan relies on microservices, as can be seen in figure 6.5. It is composed of several services, which can each be deployed independently:

- An optimization engine which performs the optimization tasks.
- A front-end Web application



(a) Visualization of optimization trajectory



(b) Visualization of optimization trajectory when noise reduction is enabled

- A back-end storage database
- A REST API enabling communications of all the services

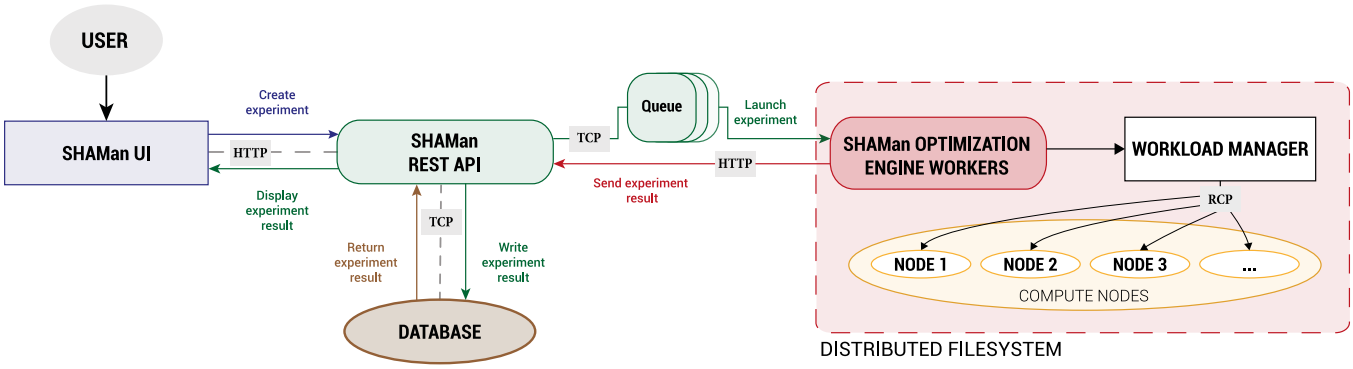


Figure 6.5: General architecture of the tuning framework

6.4.2 The optimization engine

The optimization engine is a stand-alone Python library. It is installed on a dedicated node, different from the compute nodes, so that it does not interfere with the running application, but is able to communicate with the Slurm workload manager and share the same filesystem as the compute nodes. The engine, schematically described in figure 6.6, is composed of a flexible generic black-box Python library (that can be also be used through an interactive session) and a wrapper which transforms the configurable component into a tunable black-box. The optimization engine is run by workers, triggered by a message broker system upon API call.

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 and the stop criteria described in chapter 3, as well as the noise reduction methods described in chapter 5 and pruning strategies. A Python API provides abstractions to add new optimization heuristics with minimal development cost. Adding a new optimization technique is thus straightforward and, by extension, adding new heuristics to SHAMan has a minimal development cost.

The black-box wrapper The black-box wrapper module transforms the component being tuned into a black-box. It is a Python object (called `bb_wrapper`) that configures and launches the component according to the specification given by the user in the configuration file. It builds a `.compute` method that takes as input the parameters of the component and the parameters of the experiment, edits the virtual environment, the sbatch file, and the command line, submits the job through Slurm, parses the

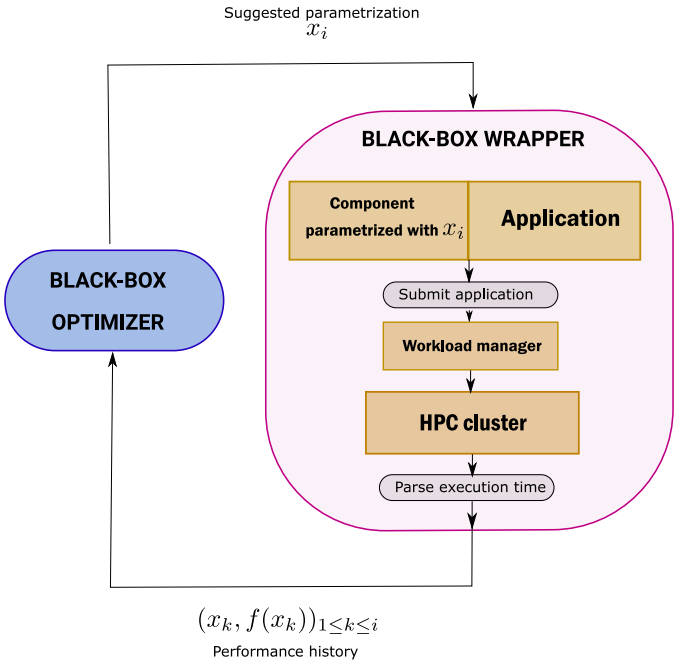


Figure 6.6: Schematic representation of the optimization engine

output of the Slurm program to get the execution time, and sends this information to the optimizer so that it can make the decision of the next parameters to use.

6.4.3 Integration of the optimizer within a Web application

As described in figure 6.5, 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 trigger of a new experiment and the workflow is described in figure 6.7. When the user triggers an optimization experiment, the REST API sends the experiment's information to the optimization workers. The optimization tasks then starts on the optimization engine. When the engine performs the tuning of the system, the results are sent to the API through HTTP calls after each application run and stored in the permanent database. If the user is currently visualizing the running experiment, a Web socket is opened by the API in order to subscribe to the collection receiving the results, so that they can be visualized in real-time.

6.4.4 Implementation choices

SHAMan uses *Nuxt.js* as a frontend framework. The optimization engine is written in Python. The database relies on the NoSQL database management system *MongoDB*. The message broker system uses Redis [13] as a queuing system, manipulated with the ARQ Python library [1]. The API is developed in Python, using the FastAPI framework. The framework is fully tested, can be fully deployed as

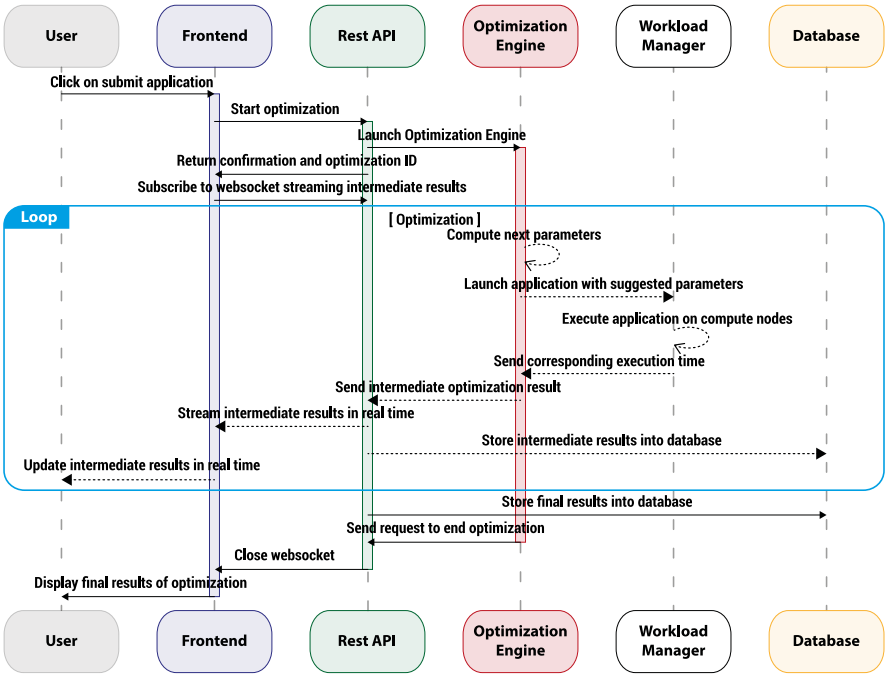


Figure 6.7: Application's workflow upon experiment submission

a stack of Docker containers [23], and is also thoroughly documented [6] [2].

6.5 Practical use-case and implementation

The use-cases given in chapter 2 and later in chapter 8 have been successfully registered and tuned using SHAMan. We detail in this section the different configuration files used to register and tune these components using SHAMan.

6.5.1 I/O accelerators

Configuration file for SRO The Small Read Optimizer (SRO), described in section 2.2, is launched through a Slurm plugin called `fastio`, as reflected in the configuration file of listings. All of its parameters, as described in table 2.1, are environment variables changed at runtime by SHAMan. The `header` empties the cache at each optimization iteration in order to ensure the independence of each optimization run. An example of configuration file for the SRO is given in listings 6.4.

Listing 6.4: Example of configuration file for the SRO

```
sro:
  plugin: fastio=yes
  header: clush -w $(hostname) -l root 'sync ; echo 3 > /proc/sys/vm/drop_caches'
  parameters:
```

```
        SRO_SEQUENCE_LENGTH:
            env_var: True
            type: int
            default: 100
        SRO_DSC_BINSIZE:
            env_var: True
            type: int
            default: 1048576
        SRO_CLUSTER_THRESHOLD:
            env_var: True
            type: int
            default: 2
        SRO_PREFETCH_SIZE:
            env_var: True
            type: int
            default: 20971520
```

Example of configuration file for SBB The Smart Burst Buffer (SBB), described in section 2.3, is launched by adding a command `#SBB` on top of the script to optimize. Each of the Smart Burst Buffer variable is passed as a variable of this command, and this is reflected in the configuration file provided in listings 6.5.

Listing 6.5: Example of configuration file for the SBB

```
sbb_small_io:
    command: "#SBB␣targets=/fs1/roberts␣flavor=small␣trash-data=1"
    parameters:
        worker-threads:
            cmd_var: True
            type: int
            default: 2
        ram-destagers:
            cmd_var: True
            type: int
            default: 2
```

```
    ram_cache_threshold:
      cmd_var: True
      type: int
      default: 10
    rdma_cq_polling_threads:
      cmd_var: True
      type: int
      default: 1
```

6.5.2 OpenMPI

Example of configuration file for tuning OpenMPI collectives The configuration of the OpenMPI collectives is done through the setting of environment variables at runtimes[1], as reflected in the SHAMan configuration file provided in listings 6.6.

Listing 6.6: Example of Open MPI configuration file

```
openmpi:
    parameters:
      OMPI_MCA_coll_tuned_allgather_algorithm:
        env_var: True
        type: int
        default: 0
      OMPI_MCA_coll_tuned_allgather_algorithm_segmentsize:
        env_var: True
        type: int
        default: 0
      OMPI_MCA_coll_tuned_allgather_algorithm_tree_fanout:
        env_var: True
        type: int
        default: 4
```

Example of parsable output for OSU benchmark The OSU benchmark [4] is optimized per size and each size gets a different runtime. To do so, a custom SHAMan parsing function must be defined. An example of a parsable output is available in listings 6.7.

[1] A thorough description of the Open MPI ecosystem is available in chapter 8

Listing 6.7: Parsable output from OSU benchmark

```
# OSU MPI Broadcast Latency Test v5.6.3
# Size          Avg Latency(us)
512                       18.71
```

6.6 Conclusion

In conclusion, we have presented in this chapter an auto-tuning framework, called SHAMan, developed to tune generic HPC components for a given application. This tool addresses some of the gaps that we found in frameworks already available in the literature, and has been made available to the Open-Source community. It has been used to provide the results described in this thesis for the tuning of I/O accelerators. We also provide in chapter 8 an application to the tuning of OpenMPI collectives.

Bibliography

[1] *ARQ*. https://arq-docs.helpmanual.io/.

[2] *Documentation of the SHAMan application*. https://shaman-app.readthedocs.io/.

[3] *mpitune*. https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/command-reference/mpitune.html.

[4] *OSU micro-benchmarks*. http://mvapich.cse.ohio-state.edu/benchmarks/.

[5] *Scikit-optimize*. https://github.com/scikit-optimize/.

[6] *The SHAMan application*. https://github.com/bds-ailab/shaman.

[7] T. AKIBA, S. SANO, T. YANASE, T. OHTA, AND M. KOYAMA, *Optuna: A next-generation hyperparameter optimization framework*, in Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining, 2019, pp. 2623–2631.

[8] J. ANSEL, S. KAMIL, K. VEERAMACHANENI, J. RAGAN-KELLEY, J. BOSBOOM, U.-M. OREILLY, AND S. AMARASINGHE, *Opentuner: An extensible framework for program autotuning*, in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, 2014, pp. 303–316.

[9] T. G. AUTHORS, *GPyOpt: A bayesian optimization framework in python*. http://github.com/SheffieldML/GPyOpt, 2016.

[10] J. BERGSTRA, D. YAMINS, AND D. COX, *Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures*, in Proceedings of the 30th International Conference on Machine Learning, vol. 28, 2013, pp. 115–123.

[11] X. BOUTHILLIER, C. TSIRIGOTIS, F. CORNEAU-TREMBLAY, T. SCHWEIZER, L. DONG, P. DELAUNAY, M. BRONZI, D. SUHUBDY, R. ASKARI, M. NOUKHOVITCH, C. XUA, S. ORTIZ-GAGNÉ, O. BREULEUX, A. BERGERON, O. BILANIUK, S. BOCCO, H. BERTRAND, G. ALAIN, D. SERDYUK, P. HENDERSON, P. LAMBLIN, AND C. BECKHAM, *Epistimio/orion: Asynchronous Distributed Hyperparameter Optimization*. https://github.com/Epistimio/orion, 2021.

[12] M. CHAARAWI, J. M. SQUYRES, E. GABRIEL, AND S. FEKI, *A tool for optimizing runtime parameters of Open MPI*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2008, pp. 210–217.

[13] M. D. DA SILVA AND H. L. TAVARES, *Redis Essentials*, Packt Publishing, 2015.

[14] 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, 2017, pp. 479–488.

[15] S. DUAN, V. THUMMALA, AND S. BABU, *Tuning database configuration parameters with iTuned*, in Proceedings of the Very Large Data Base Endowment, vol. 2, 2009, pp. 1246–1257.

[16] D. GOLOVIN, B. SOLNIK, S. MOITRA, G. KOCHANSKI, J. KARRO, AND D. SCULLEY, *Google vizier: A service for black-box optimization*, in Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017, pp. 1487–1495.

[17] J. GROHMANN, D. SEYBOLD, S. EISMANN, M. LEZNIK, S. KOUNEV, AND J. DOMASCHKA, *Baloo: Measuring and modeling the performance configurations of distributed DBMS*, in 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2020, pp. 1–8.

[18] H. HERODOTOU, H. LIM, G. LUO, N. BORISOV, L. DONG, F. CETIN, AND S. BABU, *Starfish: A self-tuning system for big data analytics*, in CIDR 2011 - 5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings, 2011, pp. 261–272.

[19] L. HERTEL, J. COLLADO, P. SADOWSKI, J. OTT, AND P. BALDI, *Sherpa: Robust hyperparameter optimization for machine learning*, in SoftwareX, vol. 12, 2020.

[20] M. JETTE, A. YOO, AND M. GRONDONA, *Slurm: Simple linux utility for resource management*, in Lecture notes in computer science, 2003.

[21] P. KOCH, O. GOLOVIDOV, S. GARDNER, B. WUJEK, J. GRIFFIN, AND Y. XU, *Autotune*, in Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018.

[22] T. T. LE, W. FU, AND J. H. MOORE, *Scaling tree-based automated machine learning to biomedical big data with a feature set selector*, in Bioinformatics, vol. 36, 2020, pp. 250–256.

[23] D. MERKEL, *Docker: lightweight linux containers for consistent development and deployment*, Linux journal, (2014).

[24] S. ROBERT, S. ZERTAL, AND P. COUVEE, *Shaman: A flexible framework for auto-tuning hpc systems*, in Revised selected papers of the 28th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2021, pp. 147–158.

[25] D. VAN AKEN, A. PAVLO, G. J. GORDON, AND B. ZHANG, *Automatic database management system tuning through large-scale machine learning*, in Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 1009–1024.

[26] B. ZHANG, D. VAN AKEN, J. WANG, T. DAI, S. JIANG, J. LAO, S. SHENG, A. PAVLO, AND G. J. GORDON, *A demonstration of the ottertune automatic database management system tuning service*, in Proceedings of the Very Large Data Base Endowment, vol. 11, 2018, pp. 1910–1913.

Chapter 7

Automatic parametrization using metadata matching

Contents

The Smart HPC Application MANager (SHAMan) auto-tuner introduced in chapter 6 stores information related to the tuning process, such as the tuned application, the optimal parameters and the associated optimal performance. In the context of offline tuning described by figure 7.1a, in order to re-use this result, the user has to manually store this information and apply it upon submission of their application to have the optimum performance. This can be done for example by editing the submission script. However, this manual process can be error prone and cumbersome. It can also cause some performance loss, because the knowledge of the similarities between applications depends on the expertise of the user on the application's behavior.

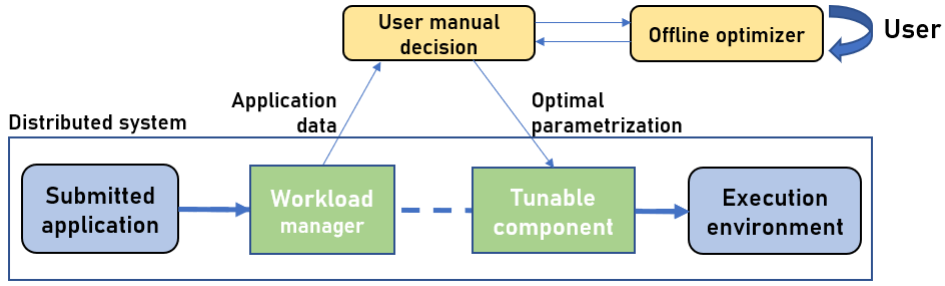
In section 1.3, we have described several alternatives to integrate online tuning into a production context, as highlighted in figure 7.1[1]. In this chapter, we describe a practical implementation of the hybrid online-offline approach represented in figure 7.1b to automatically set-up optimizable components with the best parametrization. The methodology developed in this chapter can be applied to any of the components tuned by the SHAMan auto-tuner and we choose to evaluate it on the Small Read Optimizer accelerator described in section 2.2, in a scenario representative of conditions observed in production.

The suggested metadata matching engine automatically sets the parametrization of incoming applications based on the information stored in the SHAMan database. Upon submission of a new application, an API intercepts it and matches this application with the data collected on previously optimized application, and runs the application with the optimal parametrization found for this matched application. The matching is performed on the basis of the applications metadata collected on the application within the submission environment through workload managers as well as the run-time environment. It relies on an unsupervised learning technique to match records between different data entities, called *record linkage*. The introduction of this matching pipeline is a step towards automatic system's parametrization and building a fully autonomous optimization loop, that would not require any user involvement.

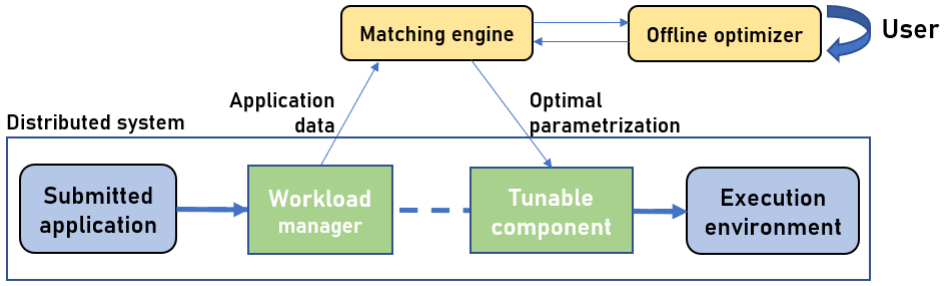
7.1 Existing online tuners

Different works in the literature are related to the prediction of optimum execution run-time using a priori application's submission information in the HPC context. In [22], Tanash et al. predict the resource consumption of an application to run it on the cluster with the optimum allocated resources,

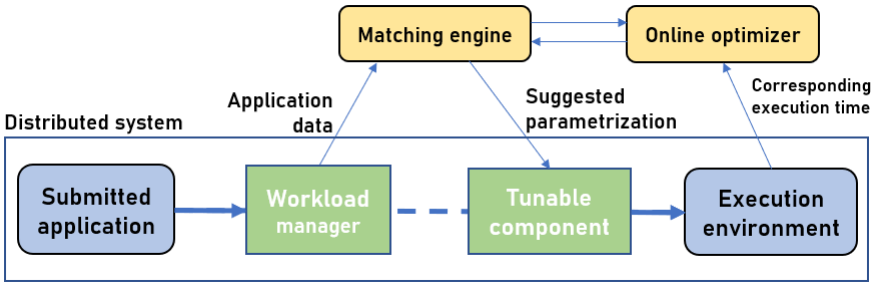
[1] This figure is also available in chapter 1



(a) Offline tuning



(b) Trade-off between offline and online tuning



(c) Online tuning

Figure 7.1: Different possible variations of online and offline tuning

instead of the often over-inflated resources given by users. To do so, they test different supervised Machine Learning models trained on Slurm's input data and predict the actual resources taken by the job.

Definition 27: Application's a priori data

Application's ***a priori* data** is data available upon the application's submission, before the application has started running.

Definition 28: Application's in situ data

Application's ***in situ* data** is data available while the application is running, usually in real-time. It includes *a priori* data, as well as data reflecting the application's activity, such as the performed I/O, the energy profile, the resource consumption ...

Another possible field of application of auto-tuning using *a priori* data is the prediction of energetic

consumption of applications, in order to create schedulers aware of the power profile of submitted applications. In [20], Saillant et al. use a regression model built on Slurm's input submission data to learn the power profile consumption of each job. They treat each field as categorical and do not take into account textual fields. While their main approach involves supervised learning and thus requires some offline training, they do take into account the possibility of using their model online by using instances re-weighting based on an exponential decay law which gives more importance to more recent data. In [4], Bugbee et al. use *a priori* data as well as *in situ* data to characterize and predict the energy consumption using different regression models (linear models, MARS[2], as well as Random Forest). In [17], Matsunaga et al. use a combination of regression models into a single one using the Predicting Query Runtime algorithm to predict the resources used by incoming applications, using the data concerning the application's metadata.

Another existing motivation for *a priori* prediction of application's behavior is the prediction of the application duration in order to improve the performance of job schedulers. It is the goal of Gaussier et al. in [11], who estimate jobs' duration by fitting a polynomial model based on the jobs' historical data. A different approach with a similar goal is described in [21] [8] and [12]. To predict the running time, they split the different applications according to *categories* or *templates*, defined by their metadata, such as the user's name or the executable name. Then, they perform the prediction within a given category using classical regression methods. In [12] and [8], Warren et al. and Downey manually define the templates, based on field knowledge as well as empirical experiments. In [21], the best performing templates are determined using automatic search algorithms, such as genetic algorithms.

7.2 Record linkage for applications matching

To match incoming applications with already known ones, we choose to use a type of unsupervised methods called *record linkage* [6]. Also known as *data matching*, it is a process to match several records corresponding to the same entity across different data sources. The first idea of Record Linkage was introduced by Halbert Dunn in 1946 [9] as a book of life for each individual and since the 1950s several research domains explored, developed and applied this technique. The main popular ones are public health science [16] [13] [18], to improve quality of clinical care and statistical studies as population census [14] to target the right development plans, governments have to anticipate. This use of Record Linkage in many domains, goes with its own continuous optimization and a sub-

[2]Multivariate adaptive regression spline

stantial work is achieved to preserve the record linkage privacy [19], cleaning and adjusting linkage errors [23]. While widely used in biostatistics and the medical field [14] [15], record linkage has to our knowledge never been used to match computer records for application re-identification.

Definition 29: Record linkage

Record linkage refers to the task of finding records in a data set that refer to the same entity across different data sources.

There are several differences between the methodology developed in this chapter and the identified relevant literature described in section 7.1. The first obvious one is the difference in terms of use-case, as the methodology we develop is centered around the automatic exploitation of knowledge collected through black-box optimization. A similar idea is mentioned as a possible perspective of the work developed in [5], but to our knowledge, was not inquired further by the author.

Another difference between the already existing solutions and ours is that we use most information available to the workload manager, including text data (such as job names, program names. . .). This data is ignored or not fully exploited by most existing solutions relying on standard Machine Learning algorithm [11][20][4][22] because of the difficulty to include this input data into this type of models. When it comes to works that group applications into categories to perform the prediction [12][8][21], they do take into account textual data but in an exact way which does not take into account similarities between information. On the contrary, our method makes the most of the diversity present in workload manager's data by using methods specific to dealing with textual data and using their similarity instead of exact comparison.

Finally, the main difference between already existing methods and ours is that, to our knowledge, each methodology developed for incoming application prediction is based on supervised learning and require beforehand training data to perform the matching. On the opposite, our approach based on record linkage is fully unsupervised and does not require to collect a representative dataset before being plugged into the submission system. Indeed, we do not require any learning phase and have no need to adapt the method for online tuning, such as the actualization of models weights described in [20]. This makes our approach more straightforward and faster to implement on unknown clusters, as we do not have to consider the model's generalization. Another advantage of using our unsupervised similarity based approach is that we can give some insight on the confidence of the match through a scoring function. This allows to set a threshold under which we do not suggest any parametrization.

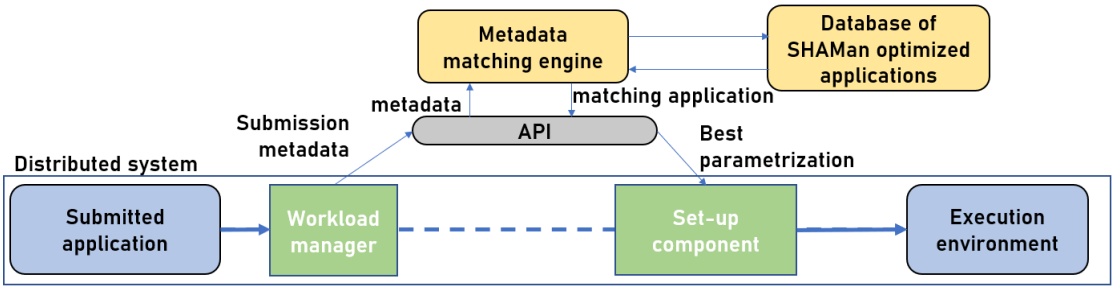


Figure 7.2: Schematic description of the matcher workflow

7.3 Tuner workflow

We present in this section the designed workflow of the auto-tuner, as well as its practical integration with the SHAMan framework. Upon submission of an application by the user, the metadata of this application is collected, using the environment variables made available by the workload manager, such as the name of the user and the name of the program, but also the requested resources, such as the number of nodes. The exhaustive list of all the data used for this study can be found in table 7.1.

Table 7.1: Metadata collected by the workload manager and used for matching

Variable name	Description	Variable type
Username	Name of the user that submitted the application	String
Jobname	Name of the job	String
Program name	Name of the running program	String
Command line	Command line used to submit the program	String
Start time	Time of submission of the application	Date
Node count	Number of nodes the application is running on	Integer

This data is then sent to the interface between the submission systems and the metadata matching engine. When this engine receives the data, it performs the matching using the records of applications that have been previously optimized by the SHAMan auto-tuner. These records contain the metadata of the optimized applications, as well as the information related to the optimization process, such as the optimal parametrization and its associated performance. The application with the best match is then sent back to the API, which sends it to the module responsible for selecting the most promising parametrization of the prefetcher. This workflow is presented schematically in figure 7.2.

In our concrete implementation of the matching pipeline architecture, the submission of the application hangs until the matching process is over, thus requiring the matching process to be efficient enough to not affect the performance of the user. The submission process becomes synchronous and this optimization is transparent to the user and his usage of the cluster.

7.4 Matching methodology

The metadata matching pipeline uses deterministic record linkage [24] to perform the matching, which is similar to performing 1-NN classification [10] with distance specific to the case of HPC metadata. The process of metadata matching is organized according to 3 steps, as represented in figure 7.3. Each step is linked to the other, hence the term of pipeline.

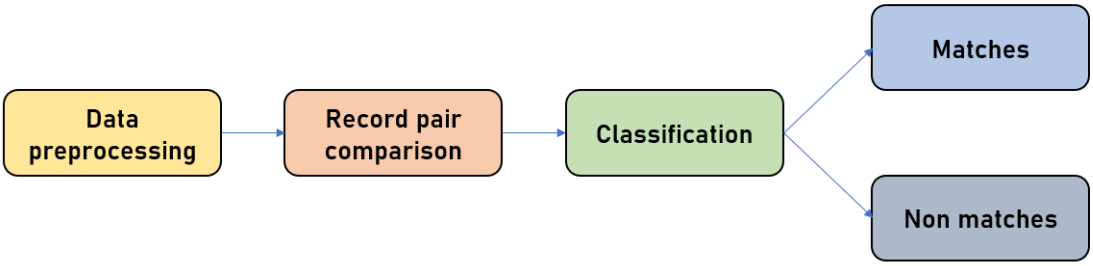


Figure 7.3: Schematic representation of the matching pipeline

7.4.1 Data preprocessing

The goal of the pre-processing step is to transform the data into a standard format so that it can be optimally compared. In our matching engine, we use 3 distinct steps of data preprocessing before performing the matching:

- **Data cleaning**: it is necessary to remove possible unnecessary characters.
- **Data segmentation**: its goal is to ensure that at the output of the cleaning block, there is no redundancy in the data.
- **Extraction of new data**: it consists in adding a possible new field to the dataset.

The cleaning and extraction process created for each of the metadata variables presented in table 7.1 are described in table 7.2.

7.4.2 Record Pair comparison

The comparison step consists in comparing 2 records field by field. As the fields presented in table 7.1, are of heterogeneous types, an adapted similarity measure must be used for each field depending on its type, in order to compute the similarity between the 2 records. At the end of this step, we obtain a vector with a similarity score for each of the already available applications for matching with the incoming application. Because we do not want a particular field to weigh more than another, each of the used similarity score gives a value located between 0 and 1.
Depending on the nature of the field, one of the similarity functions below is applied:

Table 7.2: Cleaning and extraction performed on the metadata

Variable name	**Applied cleaning**	**Extracted variables**
Username	None	
Jobname	Removal of non-alphanumeric symbols, lowercase	
Program name	Removal of non-alphanumeric symbols, lowercase	
Command line	None	Path to the program Options of the program
Start time	Projection in a trigonometric circle (depending on the periodicity of each variable)	Second Minute Hour Day Weekday IsNight
Node count	None	

- **Exact comparison** Exact comparison consists in returning a boolean if the field is exactly equal for both records, regardless of their data type (string, numeric, boolean . . .). The fields concerned by this similarity are the username as they are set by the system and are not an input from the user (*username*), and the boolean field which indicates if a job is running at night (*IsNight*).

- **Single string comparisons** To perform single string comparisons between the two string variables jobname and program name, we use the Levenshtein similarity measure, introduced in [3]. It computes the similarity between two strings as the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. The Levenshtein distance between two strings a and b is defined as:

$$sim_{lev} = \begin{cases} |a| \text{ if } |b| = 0 \\ |b| \text{ if } |a| = 0 \\ sim_{lev}(a,b) \text{ if } a[0] = b[0] \\ 1 + min \begin{cases} sim_{lev}(\text{tail}(a), b) \\ sim_{lev}(a, \text{tail}(b)) \\ sim_{lev}(\text{tail}(a), \text{tail}(b)) \end{cases} \end{cases}$$

  with the function *tail* a function returning the string without its first element. The fields compared using this similarity are the *Program Name*, as well as the *jobname*.

- **Option comparisons** To compare the options of the application between the two records, we compute for each option the ratio of values in common given for this key, and attribute the value of this ratio to the score.

When two applications have the same option name or key, we add a constant weight depending on each value to the score for each key in common in the option dictionary. This type of comparison is applied to the extracted data on the command line.

- **Multiple string comparison** Soft Tf-Idf[3] [7] consists in considering how frequently various letter combinations appear in the whole dataset, instead of just taking into account the values for the two records. It allows to downplay strings that are repeated across every record (in our case, the /home/ suffix is very common among command lines) in order to focus on the less frequent ones. The frequency of the terms present in the records are computed and then divided by their frequency over the whole dataset. We apply this similarity to the variable containing the path of the executable (*Path*), as some parts of it appear very frequently (such as the *home* path) but brings little information about the behavior of the application itself.

- **Numerical comparisons** To keep the similarity score between 0 and 1, we can only compute traditional distance measures on values already located between -1 and 1, such as the coordinates extracted from the trigonometric projection, for the seconds, minutes, hours and weekday. For these variables, we have selected the Euclidean Distance. For the other numeric measure which represents the number of nodes (*NodeCount*), we use the *absolute* similarity, which sets the distance to zero if the two elements are too far away when compared to a threshold, and otherwise to the distance between the two data points normalized by the threshold.

$$sim_{abs} = \begin{cases} 1 - \frac{|n_1 - n_2|}{d_{max}} & \text{if } |n_1 - n_2| \leq d_{max} \\ 0 & \text{else} \end{cases}$$

with:

  - n_1: the first number to compare
  - n_2: the second number to compare

  For our experiment, we set the threshold to the number of maximum of nodes available in the cluster.

The different comparisons functions applied to the different data fields are summarized in table 7.3.

7.4.3 Matcher

The goal of the matcher is to determine if two records belong to the same entity or not. To do so, we compute the average value of the similarity score, normalized by the number of extracted field per

[3] Term frequency-Inverse document frequency

Table 7.3: Comparison function applied for each of the available fields

Variable name	**Comparison function**
Username	Exact comparison
Jobname	Levenshtein similarity
Program name	Levenshtein similarity
Path	Soft TFIDF comparison
Options	Default dict comparison
Weekday	Euclidean comparison
IsNight	Exact comparison
Second	Euclidean comparison
Minute	Euclidean comparison
Hour	Euclidean comparison
Day	Euclidean comparison
Node count	Absolute comparison

original fields so that fields like the submission time do not weigh more than others because of their many extracted fields. If no record goes above a set threshold, then the application is considered to not have any match and their parametrization is set to the default one. Otherwise, the application with the highest score is returned as matching, and its optimal parametrization is used.

In our case, the weighted matcher score is:

$$\begin{aligned}\text{score} =& \frac{1}{6} sim_{\text{username}} + \frac{1}{6} sim_{\text{jobname}} + \frac{1}{6} sim_{\text{program}} \\ &+ \frac{1}{12}(sim_{\text{path}} + sim_{\text{options}}) \\ &+ \frac{1}{24}(sim_{\text{second}} + sim_{\text{minutes}} + sim_{\text{hours}} + sim_{\text{weekDay}}) \\ &+ \frac{1}{6} sim_{\text{nodeCount}}\end{aligned}$$

For our experiment, we have decided to put some even weights on every similarity and thus on every collected metadata to avoid any design bias, but in practice, it is advised to perform a careful examination of the behavior of the users on the particular cluster in order to select the weights of each similarity.

7.5 Validation of the matcher for the Small Read Optimizer

To validate the matcher and appreciate its usefulness in a conditions close to those encountered in production clusters, we built a validation environment based on a small HPC cluster composed of 5 compute nodes and 1 admin node. Three different users are created to represent several user behaviors observed from a real setting: each user launches their needed applications in their particular way.

This test scenario allows us to validate our suggested architecture and methodology in a context very close to conditions encountered in production clusters in terms of user's behaviors.

7.5.1 Cluster usage scenario

The experiment consists in creating a pool of possible applications, defined by their I/O characteristics and their names. Each one corresponds to a HPC application which I/O patterns can be altered by a different parametrization or a different input dataset. These applications are combinations of the generated I/O patterns sensitive to the impact of the Small Read Optimizer as described in section 2.2.3.

In our validation scenario, three different users are using the cluster, each with their own behavioral profile. Each user has access to every application in the defined pool and can run it on the cluster, possibly adding its particular variations (e.g. application options or input dataset). Each particular variation and its execution is called a *job*. The I/O characteristics of the application remain the same regardless of the user who picked it, but the metadata used for the application depends on the user: for example the user can choose the jobname, he can edit the program name or add some options to the command line, as well as change the topology running the nodes and thus change the I/O behavior of the application. The possible variations in the metadata are detailed in section 7.5.3.

The users can choose to run their application on its own or they can choose to optimize it using SHAMan. These applications will then be added to the SHAMan database used to perform the matching and will have their metadata stored as well as the found optimal parametrization. This validation scenario allows us to have a set of jobs owned by each different user, as well as a set of optimized job.

7.5.2 Tested applications and environment

In practice, we have designed a set of 9218 different possible applications the users can pick from. These applications are all random variations of the SRO sensitive benchmarks described in section 2.2.3. The selection of the applications per each user is done according to the discrete uniform distribution as described in table 7.4, to model different type of user behaviors. Users can draw the same application, and the optimization database is common across all user. Each user can thus benefit from the optimization of the others. In practice, we model the behavior of 3 different users as described in table 7.4.

Table 7.5 gives the final number of each application selected by each user as well as the number of optimized ones per user. Three types of user's behaviors are simulated:

1. **User 1**: this user has the smallest number of applications, as well as a high number of jobs per application set. This provides a low diversity of jobs for this user. Their number of optimization is

Table 7.4: Parametrization of the metadata matching experiment

User ID	Number of applications	Number of jobs per application	Number of optimized jobs per application
1	10	$\mathcal{U}(5, 10)$	$\mathcal{U}(1, 2)$
2	20	$\mathcal{U}(1, 5)$	$\mathcal{U}(1, 3)$
3	25	$\mathcal{U}(5, 15)$	$\mathcal{U}(2, 4)$

also low for each application, and because of the high number of jobs per application, this gives this user the smallest optimization rate.

2. **User 2**: this user has the particularity of having many applications and a low number of jobs per application. This builds a very diverse job pool for this user. Because there is a lot of diversity and a high optimization rate, this user has the highest proportion of optimized applications, as well as the smallest number of jobs.

3. **User 3**: this user has a high number of applications, as well as a high number of jobs per application and a high optimization rate. This makes it the user with the highest number of jobs. The high number of optimization also gives it a high optimization rate.

Table 7.5: Applications and experiment numbers per used ID

User ID	Number of applications	Total number of jobs	Total number of optimization
1	10	73	13
2	20	66	40
3	25	227	71

7.5.3 Variation in metadata

Each user launches an application with a different metadata, called job. The metadata that can be modified by the user are:

- **Username** The name of the user is set by the user ID using the application.

- **Jobnames** Upon submission of the application, the user can select their chosen jobname. In practice, this is highly dependent on the user behavior, his choices and the conventions of the cluster. We define two possible users behavior, observed from production datasets. Each application has a root name, and the users can randomly add a prefix and a suffix containing the selected configuration. This mimics some guidelines that have been observed on systems' running in production. For example, if we have an application called `ant`, in two thirds of the cases, an adjective randomly picked is appended as a prefix to the jobname (becoming for example `angry_ant`) and

in one third of the cases, the number of nodes is added at the end of the jobname (running on 2 nodes, `angry_ant_2`).

- **Program name** The name of the program is not affected by the user, as we found it difficult to derive a single rule from what we observed in production. Each application thus has the same generic name *sbatch*.

- **Command line** The command line is dependent on both the user, the name of the program, as well as a random timestamp associated with the time of the generation of the application. The submission command line is then: `sbatch /home/username/program_timestamp/sbatch`.

- **Start time** To not have any bias by lumping all executions of a single user together, the user running the application is randomly selected at each round.

- **Nodes configuration** Once the user has selected an application, the topology of the system is randomly drawn from a list of available compute nodes. These nodes have different names and configurations and their choices have a strong impact on the behavior of the application and thus on the impact of the auto-tuner. This configuration is reflected by the number of nodes.

7.5.4 Evaluation metrics

We evaluate and quantify the usability of our auto-tuner according to two main metrics:

1. **Impact of the auto-tuner on execution times** We compare the time spent by the application when using the parametrization selected by the matcher, against the one using the default parametrization. This quantifies the positive impact of our auto-tuner on users' usage of the cluster.

2. **Matching elapsed time** To give insight on the impact of the auto-tuner on the user workflow, we evaluate the time needed to perform the matching between the incoming application and the previously executed applications, as a function of the number of applications in the matching database. We also provide some insights in the load resilience of the suggested architecture, by testing the impact of having several users launching their application in parallel.

Additionally, we provide insight on the relevance of the matches performed by the auto-tuner and discuss how the quality of these matches impacts the performance gain as well as the matching score. We also examine the relationship between the users' behavior and the observed performance gain attributed to the tuning process.

7.5.5 Hardware and implementation

The physical implementation of the matching pipeline is written in Python. The black-box tuning is done using SHAMan. The API performing the matching uses the REST standard and is developed using the FastAPI [1] framework. The load tests are performed using the Locust framework [2]. This framework allows to perform multiple HTTP requests on REST API, simulating the load encountered by API when running in production. It provides an easy way to configure the number of users performing a request, as well as the submission frequency, and comes with a Web Interface to vizualize in real-time the system's performance.
The 4 compute nodes on which the benchmarking applications are run, consist of Intel Xeon E5-2670 cpu (16 physical cores, 32 virtual), bi-sockets, and 62 GB of DDR4 DRAM. The API runs on an isolated node with the same characteristics.

7.6 Performance of the matcher for the Small Read Optimizer

7.6.1 Impact of the auto-tuner on execution times

Over all runs we find a median improvement of 28.39% (26.06% in mean) over the 366 different jobs and 41 applications, compared to using the application with the default parametrization. The total number of applications is different from the sum of applications per user, as some users have drawn some applications in common. When removing the 150 jobs that had been directly optimized by the user, we still have a median performance improvement of 27.50% (25.47%) over applications that have never been seen by the optimizer. The accelerability potential of each job has a strong dependence on the used application, but we found that overall, on average, no application was slowed down by using the tuner. When looking at individual runs, we find that only 7 runs out of the 366 were slowed down because of our tuner, compared to the default parametrization, for a median loss of only 3% in execution time. Overall, these results highlight the performance and usefulness of our matcher when running in production.

7.6.2 Matching quality

The study of the behavior of the matcher shows that the right application is matched in 92% of the cases and the right number of nodes are matched in 56% of the cases. As described in table 7.6, a matching application and a matching number of nodes yield a better matching score and bring a better improvement in performance than applications that do not match (from 25% to 29%).

Table 7.6: Median value of the score and the auto-tuning gain

Matched application	Matched nodecount	Score	Auto-tuning gain (%)	Number of jobs
False	False	0.73	25.61	13
	True	0.74	24.73	15
True	False	0.80	25.93	148
	True	0.93	29.27	190

7.6.3 Users behavior impact

When designing our experiment, we selected 3 different users' profile. The distribution of the score and auto-tuning gain per user profile is reported in table 7.7. We find that the lower the matching score, the lower the gain brought by the auto-tuner, as exemplified by user 1.

Table 7.7: Distribution of score and auto-tuning gain per user profile

User ID	Average score	Avg. auto-tuning gain
1	0.78	23.85
2	0.87	27.40
3	0.88	26.24

The ratio of experiments per applications also has a high impact on the gain of the auto-tuner: user 2 and user 3 benefit more from the auto-tuner, compared to user 1 which has the lowest ratio of experiment and is the one with the lowest measured gain. However, the small discrepancy in terms of performance between each users can be explained by the fact that the optimization database is shared across users: each user benefits from the optimization performed by the others. Because of this, knowledge is transferred across users and even users that do not optimize a lot can benefit from this auto-tuner. This result is encouraging, as it proves that the tuner can benefit to users that do not want or cannot perform many optimizations, for example because of lack of time or resource. The solution that we propose can thus yield good results across whole clusters, as long as a small subset of users are willing to use SHAMan.

7.6.4 Elapsed time

Time to match

The time required for matching a single application to databases of different size are described in figure 7.4. The tests are performed with an API running on a server with 94GB of RAM, with an Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30GHz with 16 cores and only 4 nodes dedicated to running the matching API. We find that for a very large database of 2000 job, the submission time goes up to 20 seconds. While this response time can seem high, it has to be considered in the perspective of the HPC context: upon submission, the application is sent to the workload manager queue and can stay

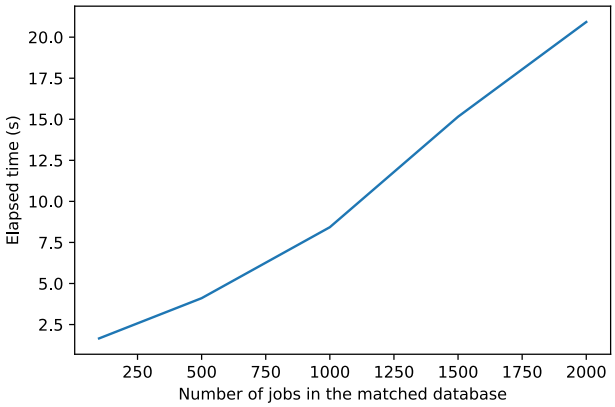


Figure 7.4: Time to match applications per database size

there for several hours until the cluster is available, especially for cluster under high traffic. Also, HPC applications often take hours to days to run.

Load tests

To test the resilience of the API to parallel submission by different users, we performed some load testing on the system running the API, for a database of 150 jobs. We perform a series of load tests simulating the behavior of users running applications on the cluster. We test a population of up to 500 users submitting an application every second. In the worst case scenario, we find a worst response time of 180 seconds, when 500 users are submitting one application every second in parallel. While the number of users stays below 200, the response time is inferior to 30 seconds (submitting a request every second), which makes the use of the auto-tuner transparent to the user even in a very high traffic scenario.

7.7 Conclusion

In conclusion, we present in this chapter a possible add-on to the SHAMan framework to automatically select parametrization upon users submission of a new application. We propose and implement an auto-tuning pipeline which relies on record linkage techniques to match an unknown incoming application with a database of previously run applications. This matcher was evaluated by auto-tuning the *Small Read Optimizer* I/O accelerator introduced in section 2.2 by comparing the performance of the applications when using the auto-tuner and without, which corresponds to using the default parametrization. We found a median performance improvement of 28% compared to using the default parametrization over a dataset representative of conditions found in production. An analysis of required elapsed times show a negligible overhead for a user submitting his job, and load tests show

that up to 200 users launching a new job every second do not have an impact on the user's experience when submitting a new job.

Bibliography

[1] *Fastapi framework*. https://fastapi.tiangolo.com/.

[2] *Locust, an open source load testing tool*. https://locust.io/.

[3] *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau?Levenshtein Distance, Spell Checker, Hamming Distance*, 2009.

[4] B. BUGBEE, C. PHILLIPS, H. EGAN, R. ELMORE, K. GRUCHALLA, AND A. PURKAYASTHA, *Prediction and characterization of application power use in a high-performance computing environment*, in Statistical Analysis and Data Mining: The ASA Data Science Journal, vol. 10, 2017, pp. 155–165.

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

[6] P. CHRISTEN, *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*, Springer, 2012.

[7] W. W. COHEN, P. RAVIKUMAR, AND S. E. FIENBERG, *A comparison of string distance metrics for name-matching tasks*, in Proceedings of the 2003 International Conference on Information Integration on the Web, 2003, p. 7378.

[8] A. DOWNEY, *Predicting queue times on space-sharing parallel computers*, in Proceedings 11th International Parallel Processing Symposium, 1997, pp. 209–218.

[9] H. DUNN, *Record linkage*, in American Journal of Public Health, vol. 66, 1946, pp. 1412–1416.

[10] E. FIX AND J. L. HODGES, *Discriminatory analysis - nonparametric discrimination: Consistency properties*, in International Statistical Review, vol. 57, 1989, p. 238.

[11] E. GAUSSIER, D. GLESSER, V. REIS, AND D. TRYSTRAM, *Improving backfilling by using machine learning to predict running times*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015.

[12] R. GIBBONS, *A historical application profiler for use by parallel schedulers*, in Job Scheduling Strategies for Parallel Processing, 1997.

[13] J. HAUKKA, R. SANKILA, T. KLAUKKA, J. LONNQVIST, N. L., T. A., K. WAHLBECK, AND T. J, *Incidence of cancer and statin usagerecord linkage study*, in International Journal of Cancer, vol. 126, 2010, pp. 279–84.

[14] M. JARO, *Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida*, in Journal of the American Statistical Association, vol. 84, 1989, pp. 414–420.

[15] M. JARO, *Probabilistic linkage of large public health data files*, in Statistics in medicine, vol. 14, 1995, pp. 491–498.

[16] C. KELMAN, J. BASS, AND D. HOLMAN, *Research use of linked health dataă a best practice protocol*, in Aust NZ Journal of Public Health, vol. 26, 2002, pp. 251–255.

[17] A. MATSUNAGA AND J. A. FORTES, *On the use of machine learning to predict the time and resources consumed by applications*, in 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010, pp. 495–504.

[18] R. MITCHELL AND J. BRAITHWAITE, *Evidence-informed health care policy and practice: using record linkage to uncover new knowledge*, in Journal of Health services Research policy, vol. 26, 2021.

[19] T. RANBADUGE AND P. CHRITEN, *A scalable privacy-preserving framework for temporal record linkage*, in Journal of Knowledge and Information Systems, vol. 62, 2020, pp. 45–78.

[20] T. SAILLANT, J.-C. WEILL, AND M. MOUGEOT, *Predicting job power consumption based on rjms submission data in hpc systems*, in High Performance Computing, 2020, pp. 63–82.

[21] W. SMITH, I. FOSTER, AND V. TAYLOR, *Predicting application run times using historical information*, in Job Scheduling Strategies for Parallel Processing, 1998, pp. 122–142.

[22] M. TANASH, B. DUNN, D. ANDRESEN, W. HSU, H. YANG, AND A. OKANLAWON, *Improving hpc system performance by predicting job resources via supervised machine learning*, in Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machine Learning, 2019.

[23] W. E. WINKLER, *Cleaning and using administrative lists: Enhanced practices and computational algorithms for record linkage and modeling/editing/imputation*, in Administrative Records for Survey Methodology, 2021.

[24] Y. ZHU, Y. MATSUYAMA, O. Y., AND S. S., *When to conduct probabilistic linkage vs. deterministic linkage? a simulation study*, in Journal of Biomedical informatics, vol. 56, 2015, pp. 80–86.

Chapter 8

Bayesian Optimization for Message Passing Interface auto-tuning

Contents

One of the advantage of black-box optimization is its ability to be applied to any tunable components. To extend our work and show the versatility of the methods developed in this thesis, we have

tested it on another kind of tunable HPC component: Message Passing Interface (MPI) runtime library. MPI is the de-facto standard for the message-passing programming model, designed for a wide range of parallel computing architectures, such as large-scale High Performance Computing (HPC) systems. It comes with hundreds of parameters, and choosing the optimal set of MPI parameters is a crucial task in order to improve the performance of HPC applications, as they spend most of their times performing communications [25]. This selection task is non-trivial as the default parametrization is often suboptimal and the optimal parametrization strongly depends on the application specific communication pattern, such as the size of messages or the number of processes involved.

Different approaches have been taken by the MPI community to tackle this problem, by using variation of the methods highlighted in section 1.2, relying on exhaustive search, analytical and machine learning models, as well as black-box optimization. However, to our knowledge, no one has tried solving it using Bayesian Optimization, through a tuning tool. Because of the versatility of Smart HPC Application MANager (SHAMan), we wanted to test if it could be successfully applied to MPI. This chapter explores the use of Bayesian Optimization with SHAMan, as described in section 3.3.2, to tune 3 parameters of 4 MPI collectives operations, in order to widen the scope of the methodology and software developed in this thesis.

This chapter begins by giving a general overview of MPI, its different collective operations, as well as the different available tunable parameters. We also give a general motivation of our study by showing both the inadequacy of default parameters and exhaustive search, as well as the main works available in the literature. The tuning experiments and the corresponding results then follow.

8.1 Tuning in the MPI ecosystem

MPI is the de-facto standard for the message-passing programming model. Its collective operations provide a standardized interface for performing data movements within groups of processes and is one of the key feature of the MPI standard.

Definition 30: Message Passing Interface

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures.

Different major implementations of this standard exist and are deployed in HPC centers, such as Open MPI [3] [15] and MPICH [1] [16]. These implementations usually adapt themselves to the underlying systems architecture to improve communications performance. A common strategy for better adapting such runtime system comes with user parametrization, allowing the specification of the network to use, the tweaking points for a communication and many other possibilities for the

various configurable parameters.

Definition 31: Open MPI

The **Open MPI** Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners.

Definition 32: MPI collectives

Collectives are operations performed by a group of threads which collectively work to perform a global communication operation. These threads can combine together to perform communication more efficiently than a single one.

For a collective operation, each implementation provide numerous algorithms to choose from, as well as different configurations for each of these algorithms.

The optimal configuration greatly depends on the size of the transmitted message [24, 31], as well as the architecture and the topology of the target platform [31], and the default parametrization is not adapted for a wide range of cases. In this section, we give some insight into the MPI ecosystem, by describing the different collectives and their tunable parameters, as well as how to parametrize them. We also justify the need for the development of an auto-tuning tool, by demonstrating the impact of the parameters of the performance and the limits of exhaustive search in terms of time to solution.

8.1.1 Open MPI Tuned collectives

The Open MPI [3] implementation is one of the most well-known open-source implementation of the MPI standard. It features a modular architecture, where different modules may implement a same set of operations and can be chosen at runtime upon user's decision. The selection of modules along with their parametrization is achieved through Modular Component Architecture (MCA) parameters, which can be provided using either configuration files, command-line arguments or environment variables.

Open MPI includes as well a selection of multiple modules for collective communication operations. The module named "tuned" [12] is a network vendor independent implementation that focuses on performance by decomposing the collective operation into point-to-point operations tailored for the underlying architecture. This module proposes, for each collective operation, a set of implementations of different algorithms such as tree-based or ring-based ones, and allows the user to tune the algorithms on, for example, the degree of tree-based algorithm, or the message segmentation size. The list of possible algorithms for each collective is detailed in table 8.1.

For the purpose of demonstrating the efficiency of black-box optimization, we have selected a subset of 4 blocking collectives to tune. They were selected amongst the most used collectives in HPC applications as highlighted by several studies [28] [25] [10], and to cover all communication patterns

Table 8.1: Collectives and their corresponding algorithms

Collectives	Algorithm name and number
Broadcast	1 - Basic linear 2 - Chain 3 - Pipeline 4 - Split binary tree 5 - Binary tree 6 - Binomial tree 7 - Knomial tree 8 - Scatter allgather 9 - Scatter allgather ring
Gather	1 - Basic linear 2 - Binomial 3 - Linear with synchronization
Reduce	1 - Linear 2 - Chain 3 - Pipeline 4 - Binary 5 - Binomial 6 - In order binary 7 - Rabenseifner
Allreduce	1 - Basic linear 2 - No overlapping (tuned reduce + tuned broadcast) 3 - Recursive doubling 4 - Ring 5 - Segmented ring

(one-to-all, all-to-one, all-to-all):

1. **Broadcast:** Broadcast is one of the collectives where one process sends the same data to all processes in a communicator. A classic use of broadcast is to send out user input to a parallel program, send out configuration parameters to all processes or simply send out the result of a local computation to multiple processes.

2. **Gather**: The gather collective takes data from several processes and gathers them to one single root process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.

3. **Reduce**: The reduce collective operation is similar to the gather operation, with the addition of a user-defined operation to apply on the collection of data. It takes an array of input elements from each process and returns an array of output elements to the root process. These output elements contain the result of a user-defined operation such as the sum or the maximum, performed on the collection of targeted data, which makes it very useful for parallel applications.

4. **Allreduce**: Many parallel applications will require accessing the reduced results across all processes rather than only the root process, and the goal of the allreduce operation is to reduce the

values and distribute the results to all processes. While being equivalent to a reduce followed by a broadcast, this explicit collective operation enables libraries to implement more efficient algorithms.

8.1.2 Tunable parameters of Open MPI

Open MPI provides a simple interface for setting up its runtime environment using the MCA parameters [5].

Definition 33: Modular Component Architecture

The Modular Component Architecture (MCA) is the backbone for much of Open MPIs functionality. It is a series of frameworks, components, and modules that are assembled at runtime to create an MPI implementation.

Several hundreds of such parameters are available for tuning, and for this study, we focus on the subset of parameters related to the `coll_tuned` component that allow to dynamically set which algorithm, fan-in/fan-out and segment size are used during a collective operation. The main reason for choosing these parameters is that they have been confirmed as having the most impact in several previous studies, especially when it comes to the algorithm [28, 31]. The tuned parameters, as well as their default values as defined in Open MPI 4.0.4, are referenced in table 8.2.

Table 8.2: Description of run-time parameters and their default values (replace * by the name of a collective)

Parameter name	Variable name	Description	Default
Algorithm	OMPI_MCA _coll_tuned_*_algorithm	Which * algorithm is used	0
Segment size	OMPI_MCA _coll_tuned_*_algorithm_segmentsize	Segmentation size in bytes used by default for * algorithms	0
Fan in out	OMPI_MCA _coll_tuned_*_algorithm_tree_fanout	Fanin/out for n-tree used for * algorithms	4

8.1.3 Setting Modular Component Architecture parameters

The variables presented in table 8.2 can be set through two different ways: as a typical MCA variable as an environment variable or in aggregated way through a configuration file. Environment variables take precedence over configuration files. The MCA variable `coll_tuned_use_dynamic_rules` should be set to `1` for it to work. Listing 8.1 is an example of such a configuration file. Comments are introduced by the # symbol.

Listing 8.1: Example of an Open MPI configuration file

```
1  # number of rules for collectives
2  # Id of the collective
# (allreduce in this case)
1  # Number of rules for nodes
8  # if nnodes >= 8
1  # number of rules for comm sizes
64 # comm size >= 64
2  # number of rules for message sizes
0 7 4 32    # msgsize algoid faninout segsize
1024 1 4 64 # msgsize algoid faninout segsize
```

This configuration file indicates that in the case of an allreduce operation, with 8 nodes or more and 64 MPI processes or more, two rules are applied: if the message size is greater than 0 bytes (and less than 1024 bytes), the 7^{th} algorithm shall be used, with a faninout value of 4 and a segsize value of 32, and if the message size is greater than 1024 bytes, the first algorithm will be used, with a faninout value of 4 and a segsize value of 64. This is a simplified example, more rules for other collectives, node counts or communicator sizes can be added. This file format allows to set a large number of rules and conditions to their application. To use this file as the Open MPI configuration file, the variables `coll_tuned_dynamic_rules_filename` and `coll_tuned_dynamic_rules_fileformat` should be respectively set to the path of the configuration file and to `1`.

8.1.4 The importance of tuning and the impracticability of exhaustive search

The importance of selecting the right parameters when running an MPI application is illustrated in figure 8.1. It shows the percentage difference in elapsed time for the OSU benchmark [4] when using the best possible parametrization found by testing every possible one, compared to the default one. Depending on the collective, the performance difference can go up to 100%, illustrating the importance of selecting the best possible parameters for a given MPI application. This gain is also highly sensitive to the message size, the collective used and the number of MPI processes used which motivates our approach of relying on black-box methods rather than on analytical ones.

The difficulty of this tuning challenge is well known across the MPI community and several studies are available regarding MPI applications tuning. Several studies confirm and further develop the results of our own study: the optimal collective parametrization depends on many factors, such as

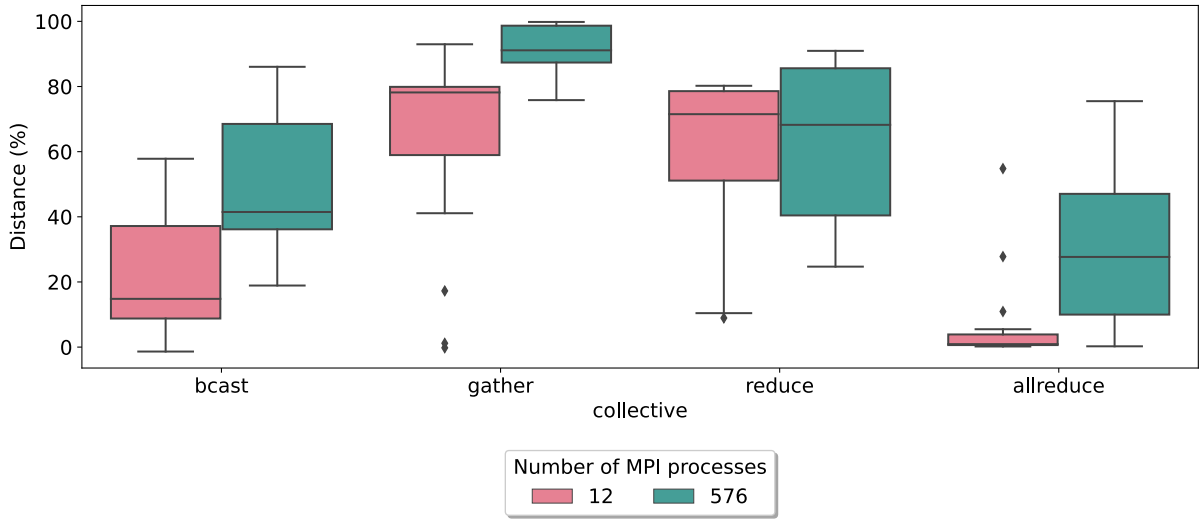


Figure 8.1: Comparison of execution time with best and default parametrizations per collective

the physical topology of the system, the number of processes involved, the sizes of the message, as well as the location of the root node [24][29]. An especially thorough analysis of the performance gap between the default parametrization and the optimum parametrization found by exhaustive search for the MPICH implementation is provided by Vadhiyar et al. in [30]. As confirmed by our own study, the default parametrization performs relatively well when the message size is small because it is well handled by the network, but as the message size gets larger, a contention phenomenon degrades the communication performance [13] [19]: for small message sizes, the number of point-to-point communication steps has to be minimized, while for larger sizes, the throughput needs to be maximized [9]. The importance of choosing the right algorithm to perform the collective operations is also emphasized in the works of Nishtala et al. in [22], as well as the works of Thakur et al. in [28]. Using parametrization adapted to the message size and the collective has thus been proven necessary to avoid this contention and performance loss.

The best parametrization illustrated in figure 8.1 was found using brute force, *i.e.* testing every single possible parametrization and running the benchmark with this parametrization. The number of tested parametrizations per collective, as well as the time required to test all these parametrizations on the hardware configuration described in subsection 8.3.3, are available in table 8.3. As we can see from this table, the tuning can go up to several days in time and makes brute force impractical on a production system, as it requires too many computing resources and user time.

We can deduce from this the strong motivation of using black-box optimization through SHAMan: finding the optimal configuration of collective is crucial for the performance of the system as the default parametrization is unsuitable for many communication problems, but exhaustive search is a very impractical way of finding it.

Table 8.3: Size of the parametric space per collective and elapsed time for an exhaustive search

Collective	Exhaustive search	Time to solution (minutes)
Allreduce	1400	504.62
Bcast	2000	5841.63
Gather	800	1063.52
Reduce	1600	638.3

8.2 Auto-tuning of MPI in the literature

Several solutions to this tuning problem have been suggested by the MPI community.

The first solution, suggested by Tu et al. in [29], consists in developing new collective and algorithms that are aware of the topology (such as the number of nodes), as well as the message size, and adapt their behavior accordingly: depending on the received messages and the system's topology, the algorithm will behave differently. While efficient, this type of methods differ radically from ours as it consists in coding a new algorithm entirely. This requires a lot of skills and expertise on the part of the user, while SHAMan has been specifically designed to be used without any knowledge on the tuned component, as highlighted in section 6.3. Our perspective is thus opposite to works suggesting the developed of new algorithms, as we place ourselves as the end users, instead of experts from the MPI field.

Another suggested solution is to use theoretical models to describe the communications, and through these models, infer the performance of the collectives and their parametrization. Several parallel communication models that predict performance of collective operations have been developed and yield good results, such as the Hockney [17], LogP [11], LogGP [6] and PLogP [21] models. Pjesivac-Grbovic et al. [24], Thakur et al. [28] and Barchet-Estefanel et al. [7] share a similar approach by using these well known models to select for each collective its optimal parametrization. In [20], Jha et al. also use these models to predict the performance of collectives and their associated algorithms. However these approaches comes with several drawbacks. Indeed, as highlighted in [24] and [20], they fail to capture some of the more complex behavior of some collective operations and do not always yield good results. They are also more rigid than our suggestion through SHAMan, as new collectives and new algorithms would need to create entirely new model. Our solution is thus more robust to future new implementations.

In the last years, a new interest of using machine learning models to predict the optimum parameters has emerged. In [23], Pellegrini et al. use machine learning models trained on metrics collected by profiling the communication behavior of MPI microbenchmarks to predict the speed-up of the application compared to the default parameters. The authors derive good results when tuning scientific benchmarks unseen by the predictor, but this method involves re-training the model each time the

architecture of the system changes. As this training phase requires a profiling tool as well as a large number of runs (the authors used 3000 instances), it is more cumbersome to deploy for the user than the method that we suggest. Another approach is developed by Hunold et al. in their works [18] and [19] which build machine learning models to predict the optimum algorithm configuration. They explore random forest in [19] and xGboost, generalized additive models, and k-nearest neighbors in [18] to respectively learn the speed-up and the running time of MPI benchmarks. While their approach does not require a profiler as they learn from the message size, the number of nodes as well as the number of processes per compute node, they only focus on predicting the optimum algorithm instead of our set of 3 parameters. Like all machine learning based approaches, they are also dependent on training offline phases which have to be repeated each time the architecture changes.

The last solution explored in the literature is the one which is closest to our suggested methodology through SHAMan. It consists in performing the tuning through empirical evaluations directly on the system, and using these evaluations to select the best performing parametrization. This approach is taken by both Vadhiyar et al. in [30] and Faraj et al. in [13]. Both authors perform an exhaustive sampling of the parametric space and reach the same conclusion as us on the impracticability of evaluating that many data points. They thus both suggest and test a black-box optimization hill-climbing algorithm, similar to simulated annealing described in section 3.3.3, to search the parametric space more efficiently. Our approach differs in two regards. The first one is obviously the selected black-box optimization algorithm, as they use variants of hill-climbing algorithms and we use the more recent Bayesian Optimization algorithm. This choice was motivated using previous results computed on I/O accelerators detailed in chapter 4, as well as after an initial study comparing the different heuristics presented in chapter 3. To our knowledge, Bayesian Optimization has never been used as a black-box algorithm to tune MPI, making this study an original one. The other difference is the size of the exhaustively tested parametric space, as we test a lot more parametrization. Indeed, in [30], the authors only test a total of 1840 parametrization, while we test a total of 5800, as detailed in table 8.3. Our study is thus more systematic and explores more thoroughly the parametric space, and the convergence speed-up we measure computed over more parametrization.

When it comes to tools available for tuning MPI, several tools have been made available by the community to perform the tuning. The standard tool used by the Open MPI community for tuning MCA parameters is OPTO [8]. The Intel MPI community shares the same approach and use the `mpitune` [2] tool. Both of these tools included within the commercial distributions are all relying on exhaustive search, and as discussed in the previous section, the tuning time associated with exhaustive search is prohibitive. Because of this, these tools have to limit the number of tested parametrization, which undermines their tuning performance compared to a more efficient search of the parametric space.

Another suggested tool is done by Sikora et al. in [26], where they extend a tool called Periscope to perform the auto-tuning of some MPI parameters using evolutionary algorithms. While the methodology can seem similar to ours, the use-case is different, as the authors do not try to tune MPI collective operations. Another tool available to the community is STAR-MPI [14], which stands for Self Tuned Adaptive Routines for MPI Collective Operations and implements some of the methods developed in [13]. It is the work the closest to ours in terms of methodology as they also use black-box optimization, but the main difference lies in the fact that STAR-MPI is intrusive in the communication process, as it delays the parametrization of the collective's algorithm during the communication, introducing an overhead. Another difference is that they only focus on the algorithm of the collective and do not take into account other parameters. Also, the experiments we conduct with SHAMan are also more thorough in terms of exploration of the parametric grid.

In the light of already existing works, our contributions to the field of MPI tuning through the SHAMan framework is the development of a tool that:

1. Performs black-box optimization instead of the exhaustive search provided by optimization tools from mainstream MPI implementations
2. Allows us to tune several MCA parameters instead of the single target value of the used algorithm
3. Does not require any *a priori* offline training phase and can adapt to any architecture
4. Enables us to provide thorough study of the improvement of Bayesian Optimization compared to on a state-of-the-art benchmark

8.3 Validation plan and experimentation context

To assess the performance of our suggested approach, we design a validation plan based on the tuning of a set of benchmarks. The evaluation criteria are two folds: the tuning should improve the system's performance compared to the default parametrization and the convergence speed compared to an exhaustive search of the parametric space.

8.3.1 Tuned applications

The tuning is performed using the OSU MPI microbenchmark suite [4], which provides tests for every collective operation. For each of the tuned collective and each tested size, we use the corresponding benchmark in the suite. To ensure stability and reduce the noise when collecting execution times, the OSU benchmark was parameterized to perform 200 warmup runs before performing the actual test.

The tuning is carried out for the four benchmarks corresponding to the collectives introduced in subsection 8.1.1, for a message size ranging from 4KB to 1MB, with a multiplicative step of 2. Two hardware configurations are selected using the 12 nodes of cluster described in section 8.3.3, to emulate two of the most common types of process placements encountered in HPC applications:

1. **One MPI process per node**: we run a single MPI process per node, for a total of 12 MPI processes. This type of setting is typical of hybrid applications relying on MPI for inter-node communications and on OpenMP for their implicit, intra-node communications.

2. **One MPI process per core**: we run 48 MPI processes per node, for a total of 576 MPI processes. This type of setting is typical of pure, MPI-only applications which rely on the MPI library for all their communications (inter-node and intra-node alike).

This results in a total of 160 SHAMan optimization experiments (4 collectives, 20 sizes and 2 different topologies). The performance metric for tuning is the time elapsed by the benchmark for the selected size of operation, as output by the OSU benchmark[1].

8.3.2 Experiment plan

To evaluate the advantage of black-box optimization compared to exhaustive search, the reference execution time is first computed which means running the different configurations of the benchmarks with the default parametrization. This default parametrization is run 100 times to account for possible noise in the collected execution time. An exhaustive sampling of the parametric space is then performed, in order to get the corresponding execution time at each possible parametrization, and select the parametrization with the minimal execution time as the optimal one, which acts as the ground truth. This ground truth is also run one hundred times for noise mitigation.

The tuning is then performed using SHAMan, with Bayesian Optimization described in 3.3.2 and configured for MPI as described in section 6.5.2. The best parametrization found by the optimization process is considered to be the best parametrization found by SHAMan and is also run one hundred times to account for noise.

8.3.3 Hardware platform and tools

All of the tests are run using Open MPI version 4.0.4, on 24 nodes platform, with 2 x AMD rome 24 cores (AMD EPYC 7402) CPUs and a Mellanox ConnectX6 HDR200 (pcie4) interconnect.

We use SHAMan for both the exhaustive search and the Bayesian Optimization.

[1] Example of such output is provided in listing 6.7 in chapter 6

8.4 Comparison of Bayesian Optimization to exhaustive sampling

The results collected during the experiment are analyzed to answer four main questions:

1. What are the improvement in performance found by using SHAMan rather than the default parametrization ?

2. How close does the parametrization found by SHAMan come to the one found through exhaustive sampling ?

3. How much time in terms of number of iterations can we gain by using SHAMan rather than exhaustive sampling ?

4. How do these results scale when increasing the number of nodes ?

As with every tuning problem, the trade-off between finding the optimal parametrization and minimizing the number of runs must be key for the interpretation of the results, and the main goal of this study is to check if SHAMan can satisfy this trade-off for the MPI problem.

8.4.1 Performance improvement compared to default parametrization

The first important result is the gain brought by using the auto-tuner rather than the default parametrization, which is represented in figure 8.2. Over all experiments, we find an average improvement of 48.4% (52.8% in median), using the best parametrization found with Bayesian Optimization. We find an average improvement of 38.42% (29% in median) for experiments with one MPI process per node and of 58.9% (65.3% in median) when using one MPI process per core, highlighting the efficiency of tuning the Open MPI parametrization instead of simply relying on the default parametrization.

The time gain brought by Bayesian Optimization varies depending on the tuned collective, as the default parametrization is more adapted than others for some. It is the case for the *allreduce* collective when running one MPI process per node, where the optimum parametrization provides a median improvement of 0.9% (1.8% on average). Other collectives have a default parametrization that is not adapted at all. It is for example the case of the gather collective with one MPI process per core, where we see an improvement of 91% in median and on average. The improvement of the default parametrization is strongly dependent on each evaluated parameter: message size, number of processes per node or collectives and is difficult to predict. This highlights the importance of tuning each configuration to get the best performance, and the need for an auto-tuning method that can be used on every architecture.

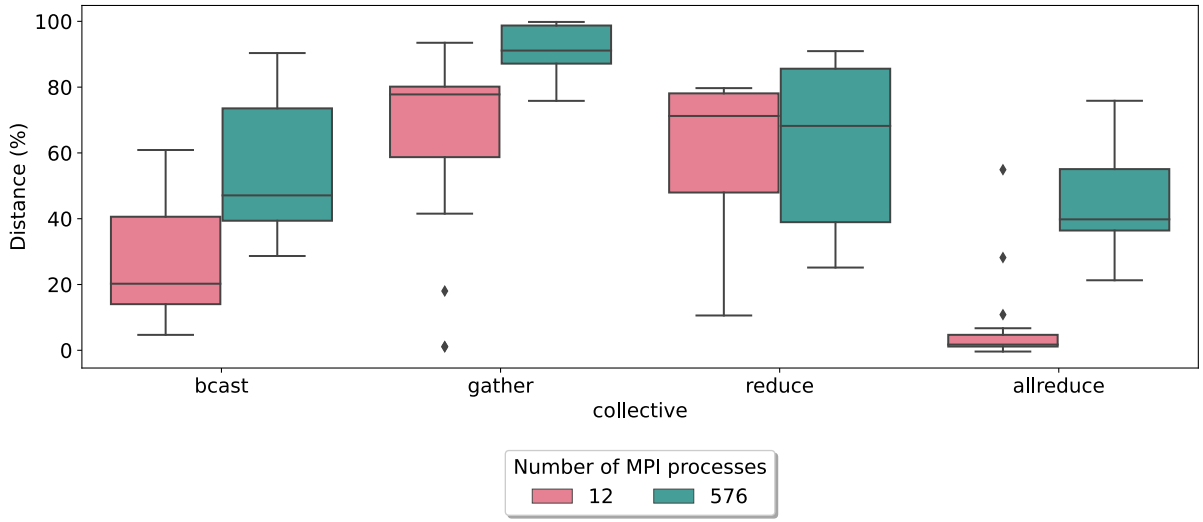


Figure 8.2: Performance gain with Bayesian Optimization compared to the default parametrization

8.4.2 Distance to exhaustive sampling

The median difference in elapsed time, along with the noise measurement, between the best parametrization found by SHAMan and the optimal parametrization found by exhaustive search is represented in table 8.4.

Table 8.4: Median difference in elapsed time and noise between best parametrization found by Bayesian Optimization and optimal parametrization

Collective	# of MPI processes	Relative difference (%)	ΔT (μs)	Noise (μs)
Allreduce	12	0.51	0.43	0.04
	576	15.28	1.05	3.81
Broadcast	12	4.79	0.32	0.26
	576	2.35	0.32	0.18
Gather	12	0.74	0.04	0.01
	576	0.00	0.02	0.00
Reduce	12	0.00	0.06	0.00
	576	0.00	0.03	0.00

Over all optimization experiments, the average distance between the optimum and the result returned by Bayesian Optimization is of 5.71 microseconds (0.04 in median) for an average noise of respectively 2.03 microseconds in mean and 0.05 microseconds in median. This means that in median, the difference between using the best parametrization of our tuner compared to the true best parametrization is imperceptible from the noise. When looking at the relative difference between the optimum and the results from Bayesian Optimization, we find an average distance of 6% (0.7% in median) between the two.

When looking at the different collectives and topologies, we find the difference between the two optimal parametrizations to be inferior to the measured noise for all collectives except for allreduce

with 576 MPI processes. When looking at each optimization problem separately, we find that for 105 optimization problems out of 160, the distance of the performance returned by Bayesian Optimization to the optimum is below the measured noise of the system. For the problems where the difference between the results returned by the tuner and the optimum cannot be explained by noise, we find a quite low average difference of 1.90 microseconds (0.18 in median). The noise difference between collectives is explained by multiple factors. Gather and reduce show low noise due to their simple communication pattern (all-to-one). On the opposite, the allreduce collective involves much more intertwined messages, which explains its higher noise and noise sensitivity. Broadcast's higher noise is explained by the best performing algorithm found (k-nomial tree) which, according to Subramoni et al. in [27], introduces some noise due the imbalanced communication pattern.

In conclusion, when comparing the results of SHAMan to the ground truth found by exhaustive sampling, we find that the tuner has been able to perform as well as the exhaustive sampler in 105 out of 160 optimization problems, and in the case it didn't, the difference was below 2 microseconds for every problem except in the case of the allreduce collective. This confirms the accuracy of our solution for optimization and makes it a satisfactory alternative to exhaustive search, especially when considering the strong improvement it brings when compared to the default parametrization.

8.4.3 Convergence speed-up

The elapsed time required to reach the optimum for the two tuning solutions and for each of the collectives and hardware configurations is reported in table 8.5.

Table 8.5: Time to solution for each heuristic and each collective

Collective	# of MPI processes	Exhaustive search (minutes)	SHAMan (minutes)	Gain (%)
Allreduce	12	52.07	4.48	**91.40**
	576	452.55	46.77	**89.66**
Bcast	12	744.10	23.65	96.82
	576	5097.53	133.25	**97.39**
Gather	12	23.45	3.42	**85.42**
	576	1040.07	77.41	**92.56**
Reduce	12	87.50	5.24	**94.01**
	576	550.80	61.58	**88.82**

With a time gain of more than 85% for each collective, we see the benefit of using guided search heuristics to explore the parametric space instead of testing every parametrization with exhaustive sampling. The time required to run all the 160 optimization experiments ranges from a total of 8048 minutes (approximately 134 hours) using brute force to 355 minutes (approximately 6 hours) using Bayesian Optimization, resulting in a total speed-up of 95%. The speed-up is relatively uniform across each collective and each topology.

The median number of iterations per collective is reported in table 8.6. The number of iterations is stable across each collective and each parametrization (mean number of approximately 30), and we can see how much guided search reduces the number of iterations.

Table 8.6: Median number of iterations performed by Bayesian Optimization compared to exhaustive search

Collective	Exhaustive search	Bayesian Optimization
Allreduce	1400	29
Bcast	2000	30
Gather	800	26
Reduce	1600	29.5

In conclusion, we are speeding-up the tuning process by 95% while maintaining an accuracy of the solution that is 6% away from to the optimal solution in average (0.7% in median), thus reaching 94% of the average improvement potential. As the main focus of the analysis is the study of the trade-off between tuning time and accuracy of the solution, we can say that SHAMan is a satisfying solution for the problem of MPI tuning.

8.4.4 Scalability study

The last axis of this study is the scalability of the suggested solution. Studying the impact of the number of nodes on the tuning time is an important task, because the number of nodes used in our study is smaller than what is usually encountered on scientific applications running in production clusters with hundreds of nodes. Because the increase in MPI processes increases the time required to process messages and thus the benchmarking time, we must study the relationship between the number of nodes and the duration of the tuning experiments. This is necessary to make sure that SHAMan can be used on production systems.

To perform this study, we run the optimization experiments on 6, 12, 18, 24 nodes, with 1 MPI process per node and only one collective. The corresponding evolution of the time to tuning for different numbers of nodes is available in table 8.7.

Number of nodes	Brute force time (minutes)	Bayesian Optimization time (minutes)
6	160.52	4.46
12	185.32	5.98
18	209.19	5.81
24	218.41	6.39

Table 8.7: Evolution of elapsed time per number of nodes

From this result, we see that the number of nodes has a non-linear impact on the convergence of the algorithms, both in the case of brute force and Bayesian Optimization. In the case of Bayesian Optimization, the elapsed time for tuning is especially stable across the number of nodes, because

of the early detection of non-promising parametrization, and gives us confidence that our suggested solution can scale well on large-scale production HPC clusters.

8.5 Conclusion

This chapter details the use of Bayesian Optimization and SHAMan for finding the optimal parametrization of MPI collective communications within its main network-agnostic component called "tuned". This component is a particularly interesting use-case for SHAMan, because the default parametrization yields poor results, and the parameters have a strong impact on the performance.
To check the relevance of our method, we peform the optimization of four MPI collective communication operations, on two different hardware topologies and for 20 different message sizes. We demonstrate that using Bayesian Optimization, we reach 94% of the average potential improvement, for a speed-up in tuning time of 95% on the overall tuning phase. Compared to default Open MPI parametrization, this leads to an average improvement of 48.4% in collective operation performance. The scalability study shows that our suggested tuner scales well and that our proposed technique would be relevant for tuning such parameters on large-scale HPC configurations.
This study confirms the versatility of SHAMan and black-box optimization for the tuning of a wide range of parametrizable components, and shows that the scope of our work can be extended to many of tunable components within the ecosystem.

Bibliography

[1] *MPICH: a high performance and widely portable implementation of the Message Passing Interface (MPI) standard*. https://www.mpich.org/.

[2] *mpitune: tunes the intel mpi library parameters for the given mpi application.* https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/command-reference/mpitune.html.

[3] *Open MPI: Open Source High Performance Computing*. https://www.open-mpi.org/.

[4] *OSU micro-benchmarks main page*. http://mvapich.cse.ohio-state.edu/benchmarks/.

[5] *Using MCA parameters with mpirun*. https://docs.oracle.com/cd/E19708-01/821-1319-10/mca-params.html.

[6] A. D. ALEXANDROV, M. IONESCU, K. SCHAUSER, AND C. SCHEIMAN, *Loggp: incorporating long messages into the logp modelone step closer towards a realistic model for parallel computation*, in SPAA '95, 1995.

[7] L. BARCHET-ESTEFANEL AND G. MOUNIÉ, *Fast tuning of intra-cluster collective communications*, in Lecture Notes in Computer Science, vol. 3241, 2004.

[8] M. CHAARAWI, J. M. SQUYRES, E. GABRIEL, AND S. FEKI, *A tool for optimizing runtime parameters of Open MPI*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2008, pp. 210–217.

[9] E. CHAN, M. HEIMLICH, A. PURKAYASTHA, AND R. VAN DE GEIJN, *Collective communication: Theory, practice, and experience*, in Concurrency and Computation: Practice and Experience, vol. 19, 2007, pp. 1749–1783.

[10] S. CHUNDURI, S. PARKER, P. BALAJI, K. HARMS, AND K. KUMARAN, *Characterization of MPI usage on a production supercomputer*, in SC'18: International Conference for High Performance Computing, Networking, Storage and Analysis, 2018, pp. 386–400.

[11] D. CULLER, R. KARP, D. PATTERSON, A. SAHAY, K. E. SCHAUSER, E. SANTOS, R. SUBRAMONIAN, AND T. VON EICKEN, *LogP: Towards a realistic model of parallel computation*, in SIGPLAN Notices, vol. 28, 1993, pp. 1–12.

[12] G. E. FAGG, G. BOSILCA, J. PJEŠIVAC-GRBOVIĆ, T. ANGSKUN, AND J. J. DONGARRA, *Tuned: An open mpi collective communications component*, in Distributed and Parallel Systems, 2007, pp. 65–72.

[13] A. FARAJ AND X. YUAN, *Automatic generation and tuning of mpi collective communication routines*, in Proceedings of the 19th Annual International Conference on Supercomputing, 2005, pp. 393–402.

[14] A. FARAJ, X. YUAN, AND D. LOWENTHAL, *Star-mpi: self tuned adaptive routines for MPI collective operations*, in Proceedings of the International Conference on Supercomputing, 2006, pp. 199–208.

[15] E. GABRIEL, G. E. FAGG, G. BOSILCA, T. ANGSKUN, J. J. DONGARRA, J. M. SQUYRES, V. SAHAY, P. KAMBADUR, B. BARRETT, A. LUMSDAINE, R. H. CASTAIN, D. J. DANIEL, R. L. GRAHAM, AND T. S. WOODALL, *Open MPI: Goals, concept, and design of a next generation MPI implementation*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2004, pp. 97–104.

[16] W. GROPP, E. LUSK, N. DOSS, AND A. SKJELLUM, *A high-performance, portable implementation of the mpi message passing interface standard*, in Parallel Computing, vol. 22, 1996, pp. 789–828.

[17] R. W. HOCKNEY, *The communication challenge for MPP: Intel paragon and meiko cs-2*, in Parallel Computing, vol. 20, 1994, pp. 389–398.

[18] S. HUNOLD, A. BHATELE, G. BOSILCA, AND P. KNEES, *Predicting MPI collective communication performance using machine learning*, in 2020 IEEE International Conference on Cluster Computing (CLUSTER), 2020, pp. 259–269.

[19] S. HUNOLD AND A. CARPEN-AMARIE, *Algorithm selection of mpi collectives using machine learning techniques*, in IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 2018, pp. 45–50.

[20] S. JHA AND E. GABRIEL, *Impact and limitations of point-to-point performance on collective algorithms*, in Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2016, pp. 261–266.

[21] T. KIELMANN, H. E. BAL, AND K. VERSTOEP, *Fast measurement of logp parameters for message passing platforms*, in Parallel and Distributed Processing, 2000, pp. 1176–1183.

[22] R. NISHTALA AND K. A. YELICK, *Optimizing collective communication on multicores*, in Proceedings of the First USENIX Conference on Hot Topics in Parallelism, 2009.

[23] S. PELLEGRINI, W. JIE, F. THOMAS, AND H. MORITSCH, *Optimizing mpi runtime parameter settings by using machine learning*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2009.

[24] J. PJESIVAC-GRBOVIC, T. ANGSKUN, G. BOSILCA, G. FAGG, E. GABRIEL, AND J. DONGARRA, *Performance analysis of MPI collective operations*, in Cluster Computing, vol. 2005, 2005.

[25] R. RABENSEIFNER, *Automatic MPI counter profiling of all users: First results on a cray t3e 900-512*, in CUG Conference Proceedings, 2004.

[26] A. SIKORA, E. CÉSAR, I. COMPRÉS, AND M. GERNDT, *Autotuning of MPI applications using ptf*, in Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications, 2016, pp. 31–38.

[27] H. SUBRAMONI, K. KANDALLA, J. VIENNE, S. SUR, W. BARTH, K. TOMKO, R. MCLAY, K. SCHULZ, AND D. PANDA, *Design and evaluation of network topology-/speed- aware broadcast algorithms for infiniband clusters*, in Proceedings of the IEEE International Conference on Cluster Computing (ICCC), 2011, pp. 317–325.

[28] R. THAKUR, R. RABENSEIFNER, AND W. GROPP, *Optimization of collective communication operations in MPICH*, in International Journal of High Performance Computing Application, vol. 19, 2005, pp. 49–66.

[29] B. TU, M. ZOU, J. ZHAN, X. ZHAO, AND J. FAN, *Multi-core aware optimization for mpi collectives*, in Proceedings - IEEE International Conference on Cluster Computing, ICCC, 2008, pp. 322–325.

[30] S. VADHIYAR, G. FAGG, AND J. DONGARRA, *Automatically tuned collective communications*, in SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, 2000.

[31] W. ZHENG, J. FANG, C. JUAN, F. WU, X. PAN, H. WANG, X. SUN, Y. YUAN, M. XIE, C. HUANG, T. TANG, AND Z. WANG, *Auto-tuning MPI collective operations on large-scale parallel systems*, in IEEE 21st International Conference on High Performance Computing and Communications, 2019, pp. 670–677.

Chapter 9

Conclusion

Contents

9.1 Summary and main findings

This thesis explores and suggests some solutions for different problems related to the tuning of HPC systems, with a special focus on the two very different I/O accelerators described in chapter 2. The chosen solutions rely on adaptive parametrization, using black-box optimization, and explore both offline and online tuning, as detailed and explained in chapter 1.

We have presented in chapter 3 three different black-box optimization heuristics used for tuning, and performed a comparative study of the behavior of each of them for the I/O accelerators and a wide range of I/O benchmarks in chapter 4. For both I/O accelerators, we have found 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. With a distance to the true minimum inferior to 4% for every application, this method provides good convergence properties and a satisfying trade-off between convergence quality and trajectory stability. We have also confirmed the robustness of these methods as the two tuned I/O accelerators operate very differently. The convergence rate inferior to 40 steps for reaching 5% of the optimal value, demonstrates as well that the

auto-tuner can operate in a sparse production environment.

In chapter 5, we have explored and discussed the impact of the noise naturally present during the optimization process of shared systems on black-box optimization heuristics. To increase their resilience to stochastic optimization, we introduce the concept of resampling and suggest three possible improvements to dynamic resampling, one of the most popular resampling technique. Over four different experiments and two I/O accelerators, we show that our solution improves the convergence of state-of-the-art dynamic resampling by 93.5% and 24.7% for respectively the SRO and the SBB accelerator, as well as speeds-up the experiment duration by 45.76% and 58.07% for these same accelerators. We also prove the importance of using noise reduction strategies whenever tuning systems running in production, as we find that using noise reduction strategies increases the found optimum by respectively 97.46% and 61.24%.

All these methods have been made available for easy use in an Open-Source software described in chapter 6, that addresses some of the gaps in the frameworks already available in the literature.

Chapter 7 explores and tests a possible add-on to the SHAMan framework to automatically select parametrization upon users'submission of a new application, using record linkage methods to match applications' metadata. Evaluated on an I/O accelerator, this matcher brings a median performance improvement of 28% compared to using the default parametrization over a dataset representative of conditions found in production. The load tests performed on the prototype also show a good resilience to parallel use by hundreds of users and proves its suitability in a production context.

Finally, chapter 8 proves the versatility of the developed methods by tuning OpenMPI collectives algorithms. We tested this method by performing the optimization of four MPI collective communication operations, on two different hardware topologies and for 20 different message sizes. We demonstrate that using Bayesian Optimization, we obtain solutions located on average at 6% from the optimum found through testing every possible parametrization by brute force, for a speed-up in tuning time of 95% on the overall tuning phase. Compared to default OpenMPI parametrization, this leads to an average improvement of 48.4% in collective operations performance.

9.2 Summary of contributions

The main contributions of this thesis can be summarized as:

(a) **To the field of software development**: The development of a generic Open Source software that bundles a wide range of black-box optimization methods for the tuning of HPC components

(b) **To the field of system optimization**: The test of black-box optimization methods for tuning three very different HPC components notoriously difficult to tune:

- A smart prefetching algorithm to speed-up pseudo-random accesses
- A Burst Buffer
- The OpenMPI collective algorithms

(c) **To the field of noisy optimization**: The suggestion of a new resampling algorithm which outperforms state-of-the-art resampling techniques for the optimization of two I/O accelerators and different noisy settings, as well as the confirmation that noise cannot be ignored when tuning shared, noisy systems

(d) **To the field of online tuning**: The suggestion, the development and the validation of a prototype for automatic parametrization of tunable systems through metadata matching

9.3 On-going works and perspectives

To continue improving and widening the scope of the work developed during this thesis, several different topics have been identified for each presented main contribution and related research field.

9.3.1 Improving SHAMan and widening its scope

On-going works

(a) **Tuning of additional systems:** The tuning of systems beyond MPI and I/O accelerators is still under investigation. A popular use-case that has been identified as a potential tuning candidate is the SAP HANA [3] database. It is an in-memory, column-oriented, relational database management system developed and marketed by SAP SE. Its primary function as the software running a database server is to store and retrieve data as requested by the applications. Several parameters can be tuned in order to adapt the system's to the current workload, including the number of threads and the MSR 0x1a4 and M0x64 parameters, used to describe the BIOS Intel prefetcher settings. These three parameters have a really strong impact on the performance of the workload and their optimal value varies depending on the use-case.

(b) **General improvement of UI:** SHAMan is in constant improvement and the main feature under development is the possibility to register a component through the Web interface, instead of through a configuration file. This will make the user interaction with the software easier and faster. It will also provide other ways to interact with the installed components, such as deletion and field modification. All the design decisions are made with an UI designer.

Short-term goals

(a) **Widening the scope of the benchmarking applications:** To display the relevance of SHAMan and the importance of tuning in the I/O community, we plan on tuning the IO-500 benchmark [11] for the SBB, which performance measure is used to rank the most powerful I/O systems in the world. Improving the current performance of Atos' burst buffer and getting a competitive results compared to other systems would validate the interest of SHAMan for the HPC community [4].

(b) **Integration and improvement of metadata matching:** The mechanism detailed in chapter 7 could be added in a production release of SHAMan. The content and results of the chapter were based on a prototype developed for this purpose, but were not integrated to the official SHAMan release. The integration of metadata matching as a full SHAMan feature would be an interesting addition to make the most of the optimization suite.

Long term goals

(a) **User management and authentication system**: Software running in production on most systems need some kind of identity and access management, through an authentication system. The integration of an identity manager, such as Keycloak [2], will enable SHAMan to be installed on production systems requiring data protection and user management.

(b) **Widening the available submission possibilities**: for now, SHAMan has been designed to work with the Slurm [9] workload manager, but other HPC systems may use a different one or none at all. Adding new submission possibilities will allow to deploy SHAMan on more systems and test the tuning of new use-cases.

9.3.2 Improving the black-box optimization process

On-going works

Tuning using other metrics: a current investigation to see if the performance of the tuning process for I/O accelerators can be improve is to use other metrics than the execution time of the application to perform the tuning. A tested metric is to use the time spent performing I/O, which is very close to the total execution time when running I/O benchmarks, but different on scientific applications which usually spend a lot of time doing computing. Another interesting metric to optimize is the I/O bandwidth, which is a metric that contributes to the overall score of the I/O 500 benchmarks [4].

Short-term goals

(a) **Speeding-up tuning through pruning strategies:** A methodology point that would need further investigating into is the behavior of the tuner when using pruning strategies. Indeed, these pruning strategies cut off some runs and prevent us from measuring the true performance corresponding to this parametrization. This can hinder the optimization process because some knowledge on the system's behavior is lost. An interesting method to deal with this type of data, called "censored" data, is survival analysis. It is a type of methods suitable for the analysis of data involving times to some event of interest [10], which in our case the time until the cancellation. The study of such methods in this context could bring some interesting improvement in terms of convergence speed, and would maximize the benefit of pruning. A good starting point can be the work of Hutter et al. in [8], who suggest a model that fills-in the truncated data with a random draw from a Gaussian Distribution in the case of Bayesian Optimization with Random Forests.

(b) **Improving the quality of the metadata matching algorithm**: further works in this area must include a wider analysis of users' behavior on production cluster. The scenario tested in this study relied on the observations made on data collected on a HPC cluster running only benchmarks. Other types of traces would widen the scope of this study and allow to study the resilience of our matching methodology to various users' behavior.

(c) **Performing more noisy experiments**: A possible improvement of the work presented in chapter 5 is to test noise with a random time of arrival rather than the fixed tested intervals. We could this way appreciate the behavior of the noise reduction resampling algorithms when faced with more uncertainty in terms of noise arrival.

(d) **Performing constrained optimization**: throughout this thesis, we have considered the parametric space as a factorial design, where every possible combination of parameters could be tested. However, field expertise and experiments have shown us that not every combination of parameters make sense on a physical level and consequently yield bad performance: for example, in the SRO accelerator, having a cluster threshold larger than the sequence length prevents the accelerator from prefetching and these parametrization are quickly discarded during the optimization process.

Long-term goals

(a) **Grey-box optimization with analytical models**: An interesting lead to pursue is the development of "grey-box" models that would leverage both the advantage of analytical models, like

those developed by Aupy et al. in [6] and [5] and Schencket al. in [12], and of black-box optimization. These analytical models can indeed be used as a primary knowledge of the performance function, for example to compute the acquisition function in the case of Bayesian Optimization, and black-box optimization will explore empirically zones that seem the most promising as determined through the analytical models. While more works exist in the state-of-the-art for Burst Buffer modelization, a similar approach can be taken for the SRO smart prefetch strategy, by building a predictive models of the performed prefetch and the corresponding cache-hits. These type of grey-box models may enable us to improve the performance of the auto-tuner and speed-up the convergence and the quality of the optimization process.

(b) **Multi-objective optimization**: Another axis of research concerns the introduction of multi-objective target measures, so that the optimizer does not only optimize the performance but also the consumed resources. This is especially interesting in the case of the SBB, where there is a trade-off between allocating the maximum number of resources and improving the performance. A multi-objective optimization target will ensure that the minimal resources are used while the performance is maximized.

9.3.3 Adding Machine Learning to the optimization process

Other perspectives that have been investigated during an internship is the possibility to leverage the information collected by the I/O monitoring system developed by Atos [1]. This system is called IO Instrumentation and collects dynamically the I/O behavior of the application as timeseries. This data is stored in a database for latter use and visualization through a Web interface. Leveraging this data as insight on the applications' behavior to add to the black-box optimization process is an interesting lead that is being explored in a research internship.

On-going works

(a) **Building an optimization recommendation system:** As discussed in previous chapters, some applications are not sensitive to the I/O accelerators, or to SHAMan (because the default parameters are already good or because there is no real sensitivity on the value of the parameters). Using the auto-tuner on these types of applications is thus a waste of time and of computing resources, that could be avoided if a Machine Learning model were able to learn beforehand which type of application can be further accelerated with black box optimization and which one can not. If proven to be efficient, a possible use of this model in production is to use it as a recommendation tool for users. Whenever the user is scrolling the list of their already run applications, the tool can suggest which application is the most relevant for optimization.

(b) **Improving the initialization plan:** I/O traces can also be used to improve the optimization process by building an initialization plan tailored to the optimized application. In the initialization plans described in section 3.2, the characteristics of the incoming applications are not taken into account and a new initialization plan is drawn for each experiment, regardless of the application profile. A Machine Learning model trained on previous experiments can be used to suggest an initialization plan based on the application's characteristics and consequently speed-up the optimization process.

Short-term goals

(a) **Grey-box model using Machine Learning:** Similarly to our suggestion with analytical models, Machine Learning models trained on already optimized applications can be used as preliminary knowledge of the performance function, and used to compute the acquisition function when doing Bayesian Optimization. The goal of the black-box optimization algorithm is then to explore the zones that were found as promising by the Machine Learning model. This type of approach can be complementary to using an analytical model as a grey-box.

(b) **Removing outlier runs using clustering methods**: Abnormal runs can happen due to the system's interference or faulty device, and can greatly impact the optimization process, as detailed in chapter 5. If the jobs behaviors are monitored, instead of using only resampling methods, we can leverage the collected metrics and use timeseries specific clustering methods on the data [7] in order to exclude faulty runs from the optimization process.

Long-term goals

Automatic parametrization using Machine Learning: Ultimately, Machine Learning models could be used to directly predict the optimum parametrization given the I/O traces of the application, removing the need for an optimization phase altogether. This would be a great achievement in terms of usefulness of our solution, but will require to deal with several difficulties, such as the gathering of a large and representative dataset.

Bibliography

[1] *IO Instrumentation*. https://atos.net/wp-content/uploads/2018/07/CT_J1103_180616_RY_F_TOOLSTOIMPR_WEB.pdf.

[2] *Keycloak*. https://www.keycloak.org/.

[3] *SAP HANA*. https://www.sap.com/products/hana.html.

[4] *Virtual institute for I/O*. https://www.vi4io.org/start.

[5] G. AUPY, O. BEAUMONT, AND L. EYRAUD-DUBOIS, *What Size Should your Buffers to Disks be?*, in International Parallel and Distributed Processing Symposium (IPDPS), 2018.

[6] G. AUPY, O. BEAUMONT, AND L. EYRAUD-DUBOIS, *Sizing and partitioning strategies for burst-buffers to reduce io contention*, in 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 631 – 640.

[7] E. ERGÜNER ÖZKOÇ, *Clustering of Time-Series Data*, InTechOpen, 2020.

[8] F. HUTTER, H. HOOS, AND K. LEYTON-BROWN, *Bayesian optimization with censored response data*, in ArXiv, vol. 1310.1947, 2013.

[9] M. JETTE, A. YOO, AND M. GRONDONA, *Slurm: Simple linux utility for resource management*, in Lecture notes in computer science, 2003.

[10] C. KARTSONAKI, *Survival analysis*, in Diagnostic Histopathology, vol. 22, 2016, pp. 263–270.

[11] J. KUNKEL, G. MARKOMANOLIS, J. BENT, AND J. LOFSTEAD, *Vi4io/io-500-dev: Zenodo citation release*. https://github.com/VI4IO/io-500-dev, 2018.

[12] W. SCHENCK, S. EL SAYED, M. FOSZCZYNSKI, W. HOMBERG, AND D. PLEITER, *Evaluation and performance modeling of a burst buffer solution*, in ACM SIGOPS Operating Systems Review, vol. 50, 2017, pp. 12–26.

Appendix A

Acronyms

API Application Programming Interface. 137, 138

HPC High Performance Computing. 13, 14, 17–21, 23, 24, 27, 33, 34, 45, 70, 72, 86, 93, 101, 124–126, 142, 146, 151, 154, 155, 159, 160, 164, 165, 173, 178, 183, 184, 186, 187, 199, 200

I/O Input/Output. 6, 9, 14, 17, 21, 33–37, 39–43, 66, 82, 86, 96, 99, 101, 129, 142, 147, 155, 160, 171, 183–186, 188, 189

LHS Latin Hypercube Sampling. 97, 100, 103

MCA Modular Component Architecture. 14, 125, 165, 167, 171

MPI Message Passing Interface. 14, 15, 20, 21, 24, 125, 164, 165, 168, 170–174, 176–178, 184, 185, 201

NFS Network File System. 70, 71, 73

REST REpresentational State Transfer. 137, 138

SBB Smart Burst Buffer. 11, 14, 34, 39–43, 70–72, 76, 79, 90, 101, 104–107, 109, 110, 113, 114, 116–119, 140, 184, 186, 188

SHAMan Smart HPC Application MANager. 75, 124, 126–130, 132–135, 137–139, 141, 146, 150, 155, 158–160, 164, 169–178, 184–186, 188, 201

SMBO Sequential Model Based Optimization. 86, 97, 100, 103

SRO Small Read Optimizer. 11, 14, 34, 35, 37, 41, 43, 70, 72, 73, 76, 80, 96, 97, 99, 103, 104, 106, 107, 109, 110, 112–114, 116–119, 139, 155, 184, 187, 188, 200

Appendix B

Glossary

Acquisition function A function which encodes for each of the parametrization of the parameter grid its potential usefulness for solving the optimization problem. 56

Application's a priori data Data available upon the applications submission, before theapplication has started running. 147

Application's in situ data data available while the application is running, usually in real-time, including a priori data, as well as data describing the application's activity (performed I/Os, the energy profile, the resource consumption . . .). 147

Black-box optimization Black-box optimization consists in optimizing a function without making any assumption, using only the previous evaluation history and within a limited budget. 46

Cooling schedule (simulated annealing) A decreasing function which computes the value of the temperature at iteration. 61

Data node A physical node which part of its cores, memory, persistent storage are dynamically allocatedto act as anI/O buffer. 40

Fakeapp A benchmarking application developed by Atos to generate complex and precise I/O patterns. 36

Fitness function (genetic algorithms) A function which associates to each individual of the population a measurement of its capacity at solving the problem at hand. In general, for optimization problems,it consists in the value of the function to optimize measured on the given solution.. 52

Genotype (genetic algorithm) The representation of a solution of an optimization problem solved by genetic algorithms in a projected space. 52

Heuristic agnostic noise reduction A type of method which consists in adding mechanisms within the optimization process to improve noise resilience, but without modifying the optimization heuristic. 88

Heuristic specific noise reduction A type of method which consists in modifying a black-box optimization heuristic to improve its resilience to noise. 88

Latin Hypercube Design An $n \times d$ matrix, in which each column is a random permutation of {1, 2,..., n}. 49

Message Passing Interface a standardized and portable message-passing standard designed to function on parallel computing architectures. 164

Model Based Optimization Consists in constructing a regression model, often called surrogatemodel or response surface model, that predicts performance and then use this model to selectthe next data point to be evaluated. 55

Modular Component Architecture the backbone for much of Open MPIs functionality. It is a series of frameworks, components, and modules that are assembled at runtime to create an MPI implementation. 167

MPI collectives operations performed by a group of threads which collectively work to performa global communication operation. These threads can combine together to perform communi-cation more efficiently than a single one.. 165

Mutation (genetic algorithm) A function which alters a genotype to turn it into another one. 54

Neighboring function (simulated annealing) A function which given the currently considered parametrization returns a parametrization adjacent on the parametric grid. 61

Open MPI operations performed by a group of threads which collectively work to performa global communication operation. These threads can combine together to perform communi-cation more efficiently than a single one.. 165

Population (genetic algorithm) A subset of solutions selected by genetic algorithms. 51

Probability of acceptance (simulated annealing) Function which computes the probability of accepting a value worse than the current state, given the current, the new value and the systems temperature. If this value is higher than a certain random thresh-old, a solution worse than the current one is accepted. 60

Pruning strategies Consists in stopping running applications if the parametrization does not seem promising compared to other tested parametrization. 128

Record linkage the task of finding records in a data set that refer to the same entity across different data sources. 149

Reproduction / Crossover (genetic algorithm) A function which merges two individuals into one which inherits characteristics from both the parents. 54

Resampling An add-on to black-box optimization which consists in the evaluation of the same parametrization several times to have a more precise idea of the performance for this parametrization when optimizing a stochastic function. 90

Selection (genetic algorithm) A function which selects two individuals in order to breed them. 53

Sequential Model Based Optimization Method that iterates between fitting a model and evaluating additional data based on this model. 55

SHAMan component configuration file YAML file describing how the component is launched and parametrized. 129

SHAMan experiment The combination of an optimizable component, a target value to measure the performance of the component, an application and a black-box optimization heuristic. 127

SHAMan experiment configuration file YAML file defining the parametrization of the black-box heuristic and the parametric optimization grid . 134

Simulated annealing An optimization heuristic inspired by statistical mechanics in thermodynamics with the statistical ensemble of the probability distribution over all possible states of a system described by a Markov chain, where its stationary distribution converts to an optimal distribution during a cooling process after reaching the equilibrium. 59

Small Read Optimizer A dynamic data preload strategy to speed-up pseudo random accesses. 35

Smart Burst Buffer Atos' implementation of a burst buffer, an I/O accelerator which consists in a large and fast intermediate layer positioned between the computing processes and the permanent storage system, to increase I/O throughput. 39

Appendix C

Related publications

Journals

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

Sophie Robert, Soraya Zertal, Grégory Vaumourin and Philippe Couvée *2021*

- Concurrency and Computation: Practice and Experience
- 10.1002/cpe.6274

SHAMan: an intelligent framework for auto-tuning HPC systems

Sophie Robert, Soraya Zertal and Philippe Couvée *2021*

- International Journal on Computational Science & Applications
- Vol. 18, No. 1, pp. 45-68, 2021

Conferences and workshops

Record linkage for auto-tuning of High Performance Computing systems

Sophie Robert, Soraya Zertal, Lionel Vincent and Philippe Couvée *2021*

- Presented at the 2021 Symposium on Intelligent and Autonomous Systems (SIAS 2021)

Using genetic algorithms for noisy systems' auto-tuning: an application to the case of burst buffers

Sophie Robert, Soraya Zertal and Grégory Vaumourin *2020*

- Presented at the 2020 International Conference on High Performance Computing & Simulation (HPCS'20)

SHAMan: a flexible framework for auto-tuning HPC systems

Sophie Robert, Soraya Zertal and Philippe Couvée *2020*

- Demonstration at the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems - 28th International Symposium, MASCOTS 2020,in Revised Selected Papers
- 10.1007/978-3-030-68110-4_10

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

Sophie Robert, Soraya Zertal and Gaël Goret *2020*

- Presented at the 2020 conference on Intelligent Systems: Theories and applications (SITA'20)
- 10.1145/3419604.3419775

Auto-tuning of I/O accelerators using black-box optimization

Sophie Robert, Soraya Zertal and Gaël Goret *2019*

- Presented at the 2019 International Conference on High Performance Computing & Simulation (HPCS'19)
- 10.1109/HPCS48598.2019.9188173
- Poster paper track
- Outstanding poster paper award

Patents

An adaptive resampling strategy for auto-tuning of noisy HPC systems

Sophie Robert, Soraya Zertal and Grégory Vaumourin *2020*

- Patent EP20306185.8

Method for optimizing execution on high-performance computing workflows

Philippe COUVÉE, Salim MIMOUNI, Sophie ROBERT, Lionel VINCENT *2020*

- Patent EP20306242.7

Méthode de réduction de l'influence du bruit dans l'inférence des paramètres du contexte d'execution d'applications HPC

Sophie Robert, Soraya Zertal and Gaël Goret *2019*

- FR1907135
- EU20182708.6
- US16/912,802

Software

SHAMan framework

Flexible auto-tuning application for finding the optimal parametrization of noisy, expensive systems

- Stochastic optimization using black-box optimization
- Front-end: Vue.js, TailwindCSS
- Back-end: FastAPI, Python
- DevOps: docker-compose, GitHub workflows
- **Github**: bds-ailab/shaman

Appendix D

Résumé en français

La plupart des logiciels des systèmes informatiques modernes sont livrés avec de nombreux paramètres configurables qui contrôlent le comportement du système et son interaction avec le matériel sous-jacent. Ces paramètres sont difficiles à régler en s'appuyant uniquement sur l'expertise des administrateurs et des utilisateurs, à cause de la large taille des espaces paramétriques et du comportement complexe et non linéaire des systèmes informatiques. De plus, la configuration optimale dépend souvent de la charge de travail en cours d'exécution sur la machine, et les paramètres doivent être modifiés à chaque variation d'environnement. Pour cette raison, les utilisateurs doivent souvent se fier aux paramètres par défaut fournis par le fournisseur et ne tirent pas parti des performances possibles de l'exécution de leur application sur un système adéquatement paramétrisé. De plus, plus ces systèmes sont complexes, plus le choix de la bonne paramétrisation devient important, car les composants interagissent les uns avec les autres de manière difficile à se représenter facilement. À mesure que l'architecture devient de plus en plus orientée service, le nombre de composants par système augmente de façon exponentielle, ainsi que le nombre de paramètres ajustables. Ce problème est particulièrement observé dans les systèmes à Haute Performance (High Performance Computing (HPC)), car les centaines d'unités matérielles assemblées pour fabriquer des supercalculateurs créent des systèmes très complexes et hautement configurables. La performance étant la préoccupation majeure dans ce domaine, chaque composant doit être correctement paramétrisé, ce qui est presque impossible à réaliser uniquement en se basant sur l'avis d'experts.

Face à l'incapacité de s'appuyer uniquement sur les utilisateurs pour prendre des décisions adéquates pour le paramétrage de systèmes informatiques complexes, de nouvelles méthodes de réglage ont émergé dans diverses communautés informatiques pour automatiser la sélection des paramètres en fonction de la charge de travail en cours d'exécution sur le système. Parce qu'elles ne nécessitent aucune intervention humaine, ces approches sont communément appelées méthodes d'*auto-*

optimisation (*auto-tuning*), un terme qui englobe un large ensemble de méthodes issues du domaine de l'optimisation et de l'apprentissage automatique. Au fil des ans, ils ont été appliqués avec succès à un large éventail de systèmes, tels que les systèmes de stockage, les systèmes de gestion de bases de données et les compilateurs. Cette thèse explore et suggère quelques solutions pour différents problèmes liés au réglage des systèmes HPC, avec un accent particulier sur deux accélérateurs E/S très différents, décrits dans le chapitre 2. Les solutions choisies reposent sur une paramétrisation adaptative, utilisant l'optimisation par méthode boîte noire, et explorent à la fois la paramétrisation hors ligne et en ligne, comme détaillé et expliqué dans le chapitre 1.

Nous présentons dans le chapitre 3 trois heuristiques différentes d'optimisation de boîte noire utilisées pour le réglage des systèmes, et proposons une étude comparative du comportement de chacune d'entre elles pour les accélérateurs E/S et une large gamme de E/S benchmarks dans le chapitre 4. Pour les deux accélérateurs E/S, nous avons trouvé que des modèles de substitution utilisant des processus gaussiens (*Gaussian Process Regression*) et la maximisation de l'espérance attendue (*Expected Improvement*) comme fonction d'acquisition offrent le meilleur compromis entre la qualité d'optimisation et la stabilité de la trajectoire. Avec une distance au vrai minimum inférieure à 4% pour chaque application, cette méthode offre de bonnes propriétés de convergence et un compromis satisfaisant entre qualité de convergence et stabilité de trajectoire. Nous avons également confirmé la robustesse de ces méthodes car les deux accélérateurs E/S optimisés fonctionnent très différemment. Le taux de convergence inférieur à 40 étapes pour atteindre 5% de la valeur optimale, démontre ainsi que "l'auto-optimiseur" peut fonctionner dans un environnement de production clairsemé.

Dans le chapitre 5, nous discutons l'impact du bruit naturellement présent lors du processus d'optimisation de systèmes partagés sur les heuristiques d'optimisation par méthode boîte noire. Pour augmenter leur résilience à l'optimisation stochastique, nous introduisons le concept de rééchantillonnage et proposons trois améliorations possibles au rééchantillonnage dynamique, l'une des techniques de rééchantillonnage les plus populaires. Sur quatre expériences différentes et deux accélérateurs E/S, nous montrons que notre solution améliore la convergence du rééchantillonnage dynamique de pointe de 93,5% et 24,7% pour respectivement le SRO et le acrshortsbb, ainsi que d'accélérer la durée de l'expérience de 45,76% et 58,07% pour ces mêmes accélérateurs. Nous prouvons également l'importance d'utiliser des stratégies de réduction du bruit chaque fois que les systèmes de réglage fonctionnent en production, car nous constatons que l'utilisation de stratégies de réduction du bruit augmente l'optimum trouvé de respectivement 97,46% et 61,24%.

Toutes ces méthodes ont été rendues disponibles pour une utilisation facile dans un logiciel *Open-Source* décrit dans le chapitre 6, qui comble certaines des lacunes des frameworks déjà disponibles dans la littérature.

Le chapitre 7 explore et teste un éventuel ajout au framework SHAMan pour sélectionner automatiquement le paramétrage lors de la soumission par les utilisateurs d'une nouvelle application, en utilisant des méthodes de couplage d'enregistrements pour faire correspondre les métadonnées des applications. Évalué sur un accélérateur E/S, ce matcher apporte une amélioration de performance médiane de 28% par rapport à l'utilisation de la paramétrisation par défaut sur un ensemble de données représentatif des conditions trouvées en production. Les tests de charge réalisés sur le prototype montrent également une bonne résilience à une utilisation parallèle par des centaines d'utilisateurs et prouvent son adéquation dans un contexte de production.

Enfin, le chapitre 8 prouve la polyvalence des méthodes développées en ajustant les algorithmes collectifs OpenMPI. Nous avons testé cette méthode en effectuant l'optimisation de quatre opérations de communication collective MPI, sur deux topologies matérielles différentes et pour 20 tailles de messages différentes. Nous démontrons qu'en utilisant l'optimisation bayésienne, nous obtenons des solutions situées en moyenne à 6% de l'optimum trouvé en testant toutes les paramétrisations possibles par force brute, pour une accélération du temps de réglage de 95% sur la phase globale de réglage. Par rapport au paramétrage OpenMPI par défaut, cela conduit à une amélioration moyenne de 48,4% des performances des opérations collectives.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Auto-optimisation de systèmes informatiques à l'aide de méthodes d'optimisation de boîte noire: une application au cas des accélérateurs E/S

Mots clés: Auto-optimisation; Optimisation de boîte noire; Système à Haute Performance;

Résumé: La plupart des composants des systèmes à Haute Performance, qu'ils soient matériels ou logiciels, sont hautement configurable, et leurs paramètres ont un impact fort sur la performance du système. Pour maximiser la performance, il faut donc trouver les paramètres les plus appropriés pour chacune des applications exécutées sur le système de calcul, mais trouver cette paramétrisation est une tâche compliquée, de par la complexité des interactions entre les éléments du système et la variété des applications que celui-ci exécute.
Dans cette thèse, nous suggérons d'utiliser des méthodes d'auto-optimisation dîtes de boîte noire, plutôt que de régler le système manuellement ou à l'aide de modèles théoriques. Nous réalisons ainsi la comparaison de trois heuristiques d'optimisation (recuit simulé, algorithmes génétiques, et optimisation bayésienne) pour auto-paramétriser deux accélérateurs d'Entrées/Sorties (E/S) pour plusieurs applications. Pour améliorer la résilience de ces algorithmes aux interférences mesurées, nous proposons un nouvel algorithme de ré-échantillonage pour l'optimisation stochastique et permettant d'obtenir une amélioration significative par rapport à l'état de l'art. Les différentes méthodes proposées sont intégrées dans un logiciel mis à disposition de la communauté dénommé SHAMan, entièrement développé durant la thèse. Aditionnellement, nous proposons un mécanisme permettant la paramétrisation automatique des applications entrantes à l'aide d'un moteur de reconnaissance de métadonnées utilisant des méthodes de dédoublonnage. Enfin, nous élargissons notre travail à d'autres composants que les accélérateurs E/S en optimisant les opérations collectives du Message Passing Interface (MPI).

Title: Auto-tuning of computer systems using black-box optimization: an application to the case of I/O accelerators

Keywords: Auto-tuning; Black-box optimization; High Performance Computing;

Abstract: Most components of High Performance Computing systems, either hardware or software, come with many tunable parameters and their parametrization can have a significant impact on their performance. For optimal performance, the most adapted parametrization for each application running on the cluster must be determined and used. However, this parametrization is difficult to find because of the complexity of the relationship between each component and the lack of insight on the systems behavior.
In this thesis, we remove the complex task of tuning the system manually or through theoretical models, by exploring auto-tuning methods relying on black-box optimization to find the systems optimal parameters. We provide a comprehensive comparison of three different black-box optimization heuristics (simulated annealing, genetic algorithms and bayesian optimization) to tune two very different I/O accelerators over several benchmarks. To improve the resilience of these heuristics to the noise caused by the shared nature of HPC systems, we suggest a new algorithm based on resampling methods for stochastic optimization, and show a significative improvement compared to the state-of-the-art. The different described methods are made available in a generic open-source software called SHAMan, entirely developed during this thesis. Additionally, we suggest a possible integration of our work for automatic parametrization of incoming applications, by using a matching pipeline on users submission and automatically parametrizing the system's at each new application's submission. Finally, we prove the relevance of black-box optimization to test yet another class of tunable component: the collective operations of Message Passing Interface (MPI).

Maison du doctorat de l'Université Paris-Saclay 2ème étage aile ouest,
Ecole normale supérieure Paris-Saclay
4 avenue des Sciences,
91190 Gif sur Yvette, France