SHAMan: a versatile auto-tuning framework for costly and noisy HPC systems

S. Robert[1], S. Zertal[2], and P. Couvée[1]

[1] Atos BDS RD Data Management
Echirolles, France
sophie.robert@atos.net
philippe.couvee@atos.net
[2] University of UPSacaly-UVSQ
Guyancourt, France
soraya.zertal@uvsq.fr

1 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. Consequently, users often have to rely on the default parameters given by the provider, and do not take advantage of the possible performance with a more appropriate parametrization of their tuned system. The more complex these systems are, the more important the tuning becomes, as the components interact with 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 HPC systems with hundreds of devices assembled to build very complex and highly configurable supercomputers. As performance is the major concern in this field, each component must be finely tuned, which is almost impossible to achieve solely through field expertise. An additional constraint is the often noisy setting, because making resources exclusive is expensive and goes against the current highly shared programmming environments. Automatic tuning methods thus must need to take into account this possible interference on the tuned application, which degrades the performance of classical auto-tuning heuristics. 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 related to 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.

In this paper, we introduce a new Open Source software, called SHAMan (**S**mart **H**PC **A**pplication **Man**ager), 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. The framework integrates three state-of-the-art black-box optimization heuristics, as well as resampling-based noise reduction strategies to deal with the interference of shared resources, and pruning strategies to limit the time spent by the optimization process. It is to our knowledge the only generalistic optimization framework specifically tailored to find the optimum parameters of configurable HPC systems, by taking into account their specific constraints.

This paper is organized as follows. We introduce in section 2 the works related to ours, and discuss the improvements provided by SHAMan compared to the state-of-the-art. Section 3 introduces the theoretical context of black-box optimization for noisy and expensive systems. In section 4 we present the different features of SHAMan and its software architecture. In section 5, we present the advantages of using SHAMan on three use-cases by tuning two different I/O accelerators and OpenMPI collectives. Finally, section 6 concludes the paper and gives insights into planned further works.

2 Related works and software

Within the HPC community, auto-tuning has gained a lot of attention for tuning particular HPC application and improve their portability across architectures [14]. Seymour et al. [36] and Knijnenburg et al. [23] 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. Menon et al. [26] 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 [28] 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. 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 [42].

In terms of auto-tuning frameworks, several 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 [12], which relies on Tree Parzen Estimators to perform the optimization. Autotune [24] is an other framework which supports several black-box optimization techniques. Scikit-Optimize [9] which supports a wide range of surrogate modeling techniques. The SHERPA [21] library provides different optimization algorithms with the possibility to add new ones. It also comes with a back-end database and a small Web Interface for experiment visualization. GPyOpt [13] is another library for users who wish to use Bayesian Optimization. Finally, the framework TPOT [25] relies on genetic algorithms for the optimization of Machine Learning pipelines. Within the MPI community, the two most famous commercical implementations come with their own tuning tool: OPTO [15] is the standard tool used by the Open MPI community for tuning MCA parameters, and similarly mpitune [6] from Intel MPI. These methods only include exhaustive grid search, making these tools slow to use for tuning expensive HPC applications. 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.

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

(a) **Versatility**: it can handle a wide range of use-cases, and new components can be registered through a generalist configuration file.
(b) **Accessibility**: the optimization engine is accessible through a Web Interface.
(c) **Optimization diversity**: as different heuristics work differently for different systems, our framework provides several state-of-the-art heuristics.
(d) **Easy extention**: the optimization engine uses a plug-in architecture and the development of the heuristic is thus the only development cost.
(e) **Integration within the hpc ecosystem**: the framework relies on the Slurm workload manager [22] to run hpc applications. The microservice architecture enables it to have no concurrent interactions with the cluster and the application itself. It is not intrusive allowing users to launch applications on their own. Also, no privileged rights are required to use the software.
(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) **Integration of noise reduction strategies**: because of the highly dynamic nature and the complexity of applications and software stacks, running in parallel on shared ressources are subject to many interference, 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) **Integration of pruning strategies**: runs with unsuited parametrization are aborted, to speed-up the convergence process.

3 Theoretical background

SHAMan relies on black-box optimization, which 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 by the optimization loop in figure 1.

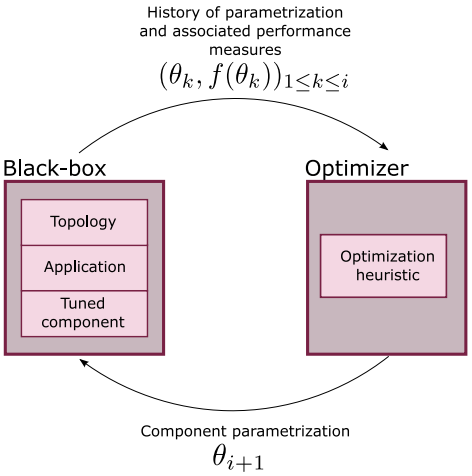

Fig. 1: Schematic representation of the optimization loop, without

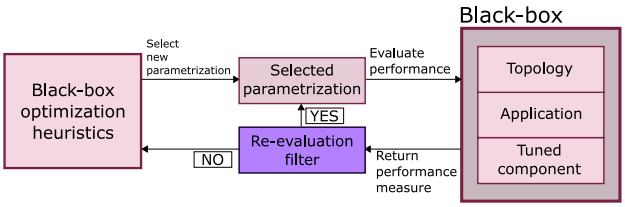

Fig. 2: Schematic representation of the optimization loop, with noise reduction through resampling

3.1 An overview of the optimization loop

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

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

- f the function to optimize
- P the parameter space
- ϵ a convergence criterion between the found and the estimated minimum

Every black-box optimization process starts with the selection of the initial parameters for the algorithm. An acceptable initialization starting plan should respect two properties [19] [41]: the space's constraints and the non-collapsible property. The space constraints are shaped by the possible values that can be taken by the parameters. The non-collapsible property specifies that no parametrization can have the same value on any dimension. A design plan respecting this constraint is called a *Latin Hypercube Design* (LHD)[19]. The next step consists in a feedback loop, which iteratively selects a parametrization, evaluates the black-box function at this point and selects accordingly the next data point to evaluate. The higher procedure for searching an optimal solution in a parametric space is called an optimization *heuristic*. There are many black-box heuristics, and we have implemented the following set in our optimization engine because of their simplicity of implementation and their proven efficiency for HPC systems' tuning. A detailed motivation and description of our implementation can be found in [31] and [32].

- **Surrogate models**: Surrogate modeling consists in using a regression or interpolation technique over the currently known surface to build a computationally cheap approximation of the black-box function and to then select the most promising data point in terms of performance on this surrogate function by using an acquisition function.

- **Simulated annealing**: the simulated annealing heuristic is a hill-climbing algorithm which can probabilistically accept a solution worse than the current one.
- **Genetic algorithms**: Genetic algorithms consist in selecting a subset of parameters, among the already tested parametrizations, considering the objective value of each parametrization, and then combining them to create a new parametrization.

3.2 Stop criteria

When running the optimization algorithm, the easiest stop criterion is based either on a budget of possible steps (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. Using other stop criterion to speed-up the convergence process, while still providing a good solution is thus essential for tuning expensive systems, and SHAMan integrates two different stop criteria:

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 [45] 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 either:

- 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. Because of this, SHAMan integrates two other criteria based either on the value of the performance function or the value of the tested parameters:

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 variations of the criteria are available in our framework:

- **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.

3.3 Resampling for noisy systems

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 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.

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 $(\theta_i \in \Theta, F(\theta_i) \in R)_k$, for the optimization trajectory at step k, and outputs a boolean on whether or not the last parametrization should be re-evaluated. This filter can be integrated for both initialization draws and exploitation draws, or only for exploitation 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. Resampling is a trade-off between having a better knowledge of the space and waste some computing times on re-evaluation. We present here two of the most popular resampling algorithms in order to efficiently reevaluate a parametrization and a more exhaustive description is proposed in [37]. Three noise strategies are available in the framework:

- **Static resampling**: re-evaluates each parametrization for a fixed number of iterations.
- **Standard Error Dynamic Resampling** [18]: re-evaluates the current parametrization until the standard error of the mean falls below a set threshold.
- **An improved noise reduction algorithm**: a complex decision algorithm for re-evaluating the current parametrization [32].

3.4 Pruning of expensive systems

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

4 Software architecture and features

SHAMan (Smart HPC Applications Manager) 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) to select the next most appropriate parametrization until the stop criterion is reached.

4.1 Terminology

Throughout this section, we will use the following terms:

- **Component**: the configurable component which optimum parameters must be found.
- **Target value**: the measurement that needs to be optimized.
- **Parametric grid**: the possible parametrization defined as (minimum, maximum, step value).
- **Application**: a program that can be run on the clusters' nodes through Slurm to be tuned.
- **Budget**: the maximum number of evaluations 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.

4.2 Optimization and vizualization procedure

The main features of SHAMan are the possibility to:

Declare a new configurable component and register it for later optimization Running the command `shaman-install` with a YAML file describing a component registers it to the application and makes it possible to be optimized. This file must describe how the component is launched and declares its different parameters and how they must be used to parametrize it. After the installation process, the components are available for optimization in the launch menu.

Design and launch an experiment through a Web interface or through a command line interface The main way is to launch the experiment through the Web interface via the menu. The user has to configure the black-box by:

1. Writing an application according to Slurm sbatch format.
2. Selecting the component and the parametric grid through the radio buttons.
3. Configuring the optimization heuristic, chosen freely among available ones. Resampling parametrization, stop criterion and pruning strategies can also be activated.
4. Selecting a maximum number of iterations and the name of the experiment.

The optimization process will begin to run and its information will be available in the exploring section of the Web application.
Another way to use SHAMan is to use it directly through a command line interface. It allows more flexibility of the different features and requires the same information as the Web interface.

Visualize data and results of finished or running experiments 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. Figure 3 and 4 show the tunes performance without any aggregation, then with it if the noise reduction is enabled.

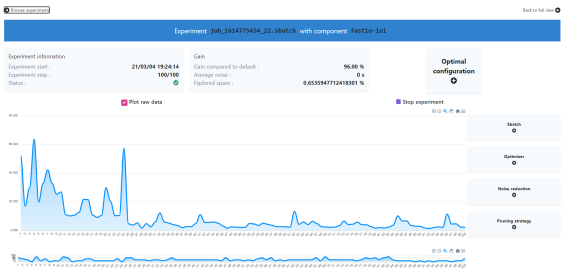

Fig. 3: Visualization of an optimization trajectory

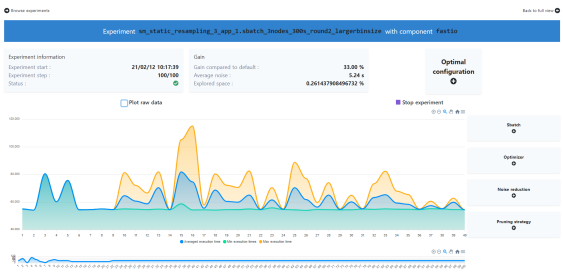

Fig. 4: Visualization of an optimization trajectory when noise reduction is enabled

4.3 Software architecture

The architecture of SHAMan relies on microservices, as can be seen in figure 5 and detailed in [33]. 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 back-end storage database
- A rest api enabling communications of all the services

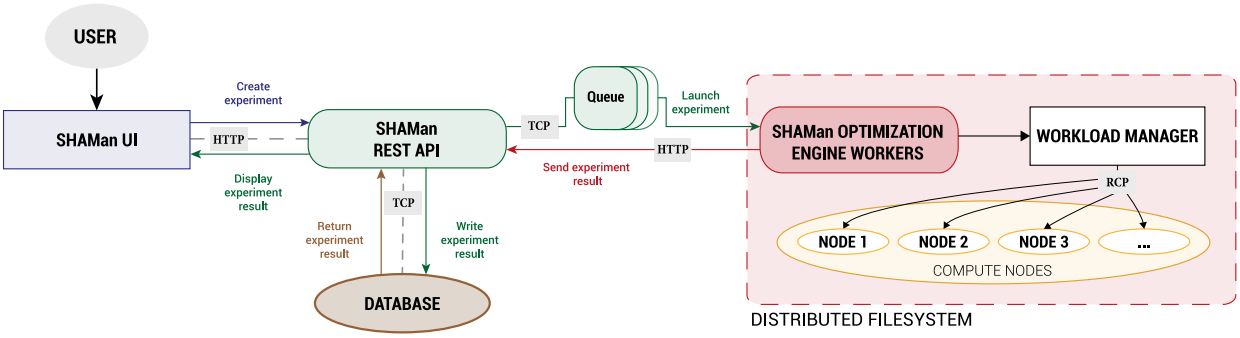

Fig. 5: General architecture of the tuning framework

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 [17] 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 a stack of Docker containers [27]. The code is available on Github [10] and thoroughly documented [3].

5 Use-cases and results

In some of our previous works, we have shown the efficiency of SHAMan on two different use-cases belonging to I/O accelerators: a smart prefetching strategy and a burst buffer [34] [35]. To further prove the versatility of SHAMan, we tackle another difficult to tune HPC component: MPI collectives. We begin this section by summarizing the main results of our previous experiments, and then introduce the new results provided by our experiment on MPI collectives.

5.1 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, which can slow down I/O intensive applications. Indeed, on large supercomputers, the many compute nodes performing reads or writes can stress the storage bay and 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. The link between the compute and the storage node can become saturated, which slows down the application. To mitigate these problems, several I/O accelerators have been developed over the years, and we focused specifically with SHAMan on tuning two commercial implementations of I/O accelerators: a pure software one called smart read optimizer [11] and a mix of software and hardware one called smart burst buffer [2]. Because these I/O accelerators come with many parameters, they are difficult to tune, and operate very differently which make them good use-cases to demonstrate the versatility of SHAMan and black-box optimization.
For both I/O accelerators, we performed a comparative study of the impact of each black-box optimization heuristic with SHAMan and showed that surrogate models offer the best trade-off between optimization quality and stability of the trajectory, and outperforms by far a random sampler.

As displayed in figure 6, we show that with a distance to the true minimum inferior to 4% for every application, our auto-tuner exhibits good convergence properties. As we have found convergence rate inferior to 40 steps for reaching 5% of the optimal value, we have also demonstrated that the auto-tuner can operate in a sparse production environment for expensive systems.

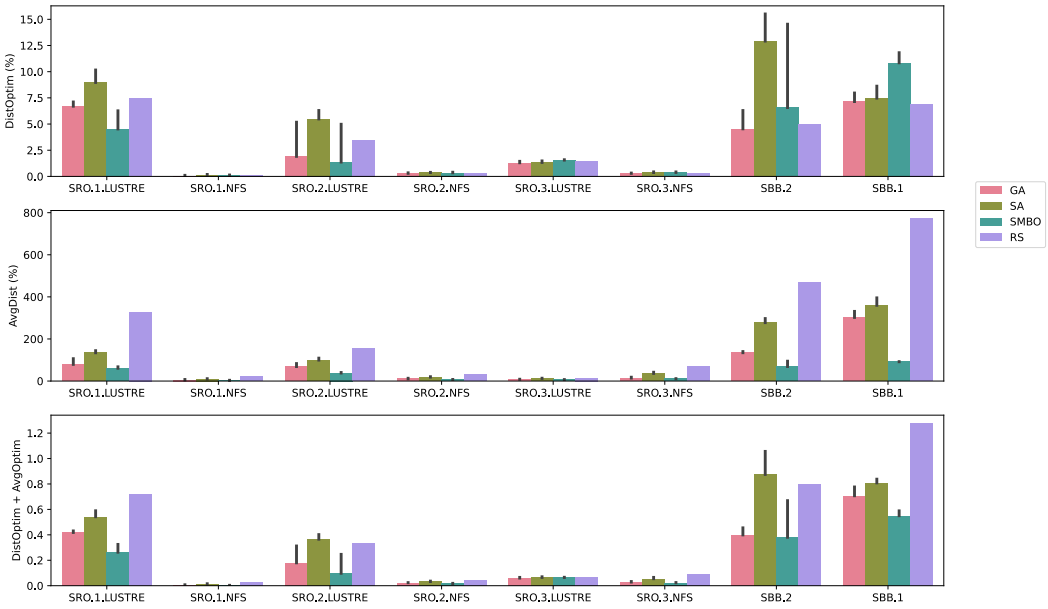

Fig. 6: Best values for DistOptim, AvgDist and the sum of both for every heuristic

5.2 Tuning MPI collectives

MPI is a standardized and portable message-passing standard designed to function on parallel computing architectures through many implementations as Open MPI [7] and MPICH [5]. Its collective operations provide a standardized interface for performing data movements within groups of processes and come with several tunable parameters. The optimal configuration greatly depends on the size of the transmitted message [30], as well as the architecture and the topology of the target platform [44], and the default parametrization is not adapted for a wide range of cases. In particular, the Open MPI [7] implementation features a modular architecture and the selection of modules along with their parametrization is achieved through MCA parameters, which can be provided using either configuration files, command-line arguments or environment variables. This implementation is the one we will be focusing on, by considering the subset of parameters related to the `coll_tuned` component that allow to dynamically set: (1) **the algorithm**; (2) **fan-in/fan-out**; (3) the **segment size**. The main reason for choosing these parameters is that they have been confirmed as having the most impact in several previous studies [39].

The importance of tuning MPI collectives The importance of this tuning challenge is well known across the MPI community and several studies confirm and further develop the results of our own study: the optimal collective parametrization depends on many factors, such as 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 [30][40]. 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 available in [43], and the importance of choosing the right algorithm to perform the collective operations is emphasized in [29]. Using parametrization adapted to the message size and the collective is thus necessary to maximize system's performance. The easiest solution for tuning is to rely on brute force, *i.e.* testing every single possible parametrization and running the benchmark with this parametrization, but this can cause the tuning process to go up to several days in time. Brute force is thus impractical on a production system, as it requires too many computing resources and user time.
All these reasons show the relevance of using black-box optimization through SHAMan: finding the

optimal configuration of MPI 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.

Experiment plan For the purpose of demonstrating the efficiency of SHAMan in the case of MPI tuning, we have selected a subset of 4 blocking collectives to tune amongst the most used ones in HPC applications [39] [16], and to cover all communication patterns (one-to-all, all-to-one, all-to-all): (1) **Broadcast**, (2) **Gather**, (3) **Reduce** and (4) **Allreduce**.

The tuning is performed using the OSU MPI microbenchmark suite [8], 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 tuned message size range from 4KB to 1MB, with a multiplicative step of 2. Two hardware configurations are selected using the 12 nodes of a cluster, to emulate two of the most common types of process placements encountered in HPC applications: (1) **One MPI process per node**, for a total of 12 MPI processes, as 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**, for a total of 576 MPI processes, 48 MPI processes per node, which 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.

To provide a thorough analysis of the advantage of black-box optimization compared to exhaustive search, the reference execution time is first computed using the default parametrization which is run 100 times to account for possible noise in the collected execution time. An exhaustive sampling of the parametric space is then performed to select the parametrization with the minimal execution time as the optimal one, which acts as the ground truth. This ground truth is also run 100 times for noise mitigation.

SHAMan parametrization The tuning is then performed with SHAMan, using Bayesian Optimization specifically configured for MPI. The initialization plan is composed of 10 parametrization. The selected maximum number of iterations is set to 150 and the stop criterion is improvement based: we choose to stop the optimization process if the best execution time over the last 15 iterations is less than 1% better than the currently found minimum. 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.

Main performance gains

Improvement compared to default The first important result is the gain brought by using the autotuner rather than the default parametrization, which is represented in figure 7. 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 SHAMan varies depending on the tuned collective, as the default parametrization is more adapted than others for some collectives. 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 compared to 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

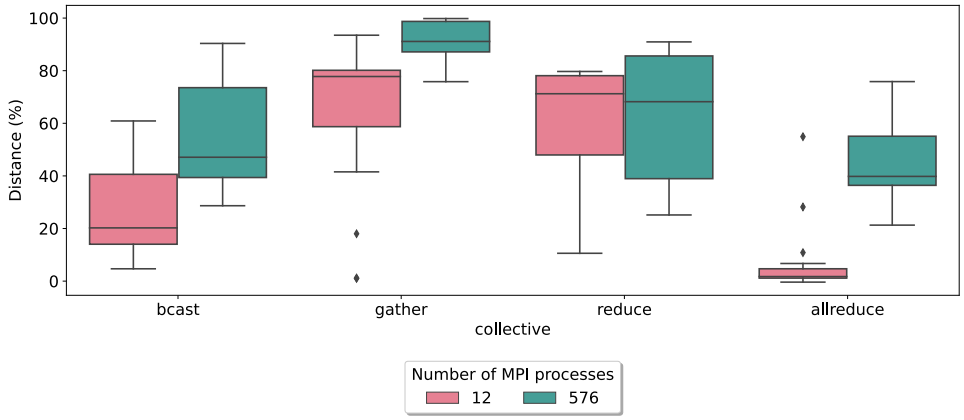

Fig. 7: Performance gain with Bayesian Optimization compared to the default parametrization

of tuning each configuration to get the best performance, and the need for an auto-tuning method that can be used on every architecture.

Tuning quality compared 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 1. Over all optimization experi-

Table 1: 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

ments, 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 [38], introduces some noise due the imbalanced communication pattern.

Tuning speed compared to exhaustive sampling The elapsed time required to reach the optimum for the two tuning solutions and for each of the collectives and configurations is reported in table 2.

Table 2: 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 with SHAMan 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.

Overall experiments, 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. 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. This study thus 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 HPC ecosystem.

6 Conclusion

In this paper, we suggest an OpenSource auto-tuning framework, called SHAMan (**S**mart **H**PC **A**pplication **M**anager) for tuning noisy and expensive systems, which addresses some of the gaps in the tuning frameworks already present in the literature. While some of our previous works already showed the strong performance of SHAMan on I/O accelerators, we added another use-case to further prove its universality, by tuning MPI collectives. When 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 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. In the near future, we intend to consider additional parameters than the topology and the message size to refine our optimization. Also, we identified additional systems to tune beyond these three cases, such as the MSR parameters of the SAP HANA database which will further extend SHAMan's scope. Regarding the improvement of the optimization methods, we plan on investigating 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. We intend to apply survival analysis to deal with this "censored" data in order to speed up even more SHAMan's convergence speed.

7 Acknowledgments

This work has been partially funded by the IO-SEA project [4], funded by the European High-Performance Computing Joint Undertaking (JU) and by BMBF/DLR under grant agreement No

955811. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, the Czech Republic, Germany, Ireland, Sweden and the United Kingdom.

-–

References

1. ARQ. https://arq-docs.helpmanual.io/.
2. 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.
3. Documentation of the SHAMan application. https://shaman-app.readthedocs.io/.
4. IO-SEA. https://iosea-project.eu.
5. MPICH: a high performance and widely portable implementation of the Message Passing Interface (MPI) standard. https://www.mpich.org/.
6. mpitune. https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/command-reference/mpitune.html.
7. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/.
8. OSU micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.
9. Scikit-optimize. https://github.com/scikit-optimize/.
10. The SHAMan application. https://github.com/bds-ailab/shaman.
11. Tools to improve your efficiency. https://atos.net/wp-content/uploads/2018/07/CT_J1103_180616_RY_F_TOOLSTOIMPR_WEB.pdf.
12. Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, pages 2623–2631, 2019.
13. The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt, 2016.
14. P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, 2018.
15. Mohamad Chaarawi, Jeffrey M. Squyres, Edgar Gabriel, and Saber Feki. A tool for optimizing runtime parameters of Open MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 210–217, 2008.
16. Sudheer Chunduri, Scott 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*, pages 386–400, 2018.
17. Maxwell Dayvson Da Silva and Hugo Lopes Tavares. *Redis Essentials*. Packt Publishing, 2015.
18. 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*, volume 2, pages 1254–1261, 2004.
19. K. T. Fang, R. Li, and A. Sudjianto. *Design and Modeling for Computer Experiments (Computer Science & Data Analysis)*. Chapman & Hall/CRC, 2005.
20. Ahmad Faraj and Xin Yuan. Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 393 – 402, 2005.
21. Lars Hertel, Julian Collado, Peter Sadowski, Jordan Ott, and Pierre Baldi. Sherpa: Robust hyperparameter optimization for machine learning. In *SoftwareX*, volume 12, 2020.
22. M. Jette, A. Yoo, and M. Grondona. Slurm: Simple linux utility for resource management. In *Lecture notes in computer science*, 2003.
23. P. Knijnenburg, T. Kisuki, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24:43–67, 2003.
24. Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. Autotune. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.
25. Trang T Le, Weixuan Fu, and Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a feature set selector. In *Bioinformatics*, volume 36, pages 250–256, 2020.
26. Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. Auto-tuning parameter choices in hpc applications using bayesian optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 831 – 840, 2020.
27. Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, (239), 2014.

28. T. Miyazaki, I. Sato, and N. Shimizu. Bayesian optimization of hpc systems for energy efficiency. In *High Performance Computing*, pages 44–62, 2018.
29. Rajesh Nishtala and Katherine A. Yelick. Optimizing collective communication on multicores. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, 2009.
30. Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham Fagg, Edgar Gabriel, and Jack Dongarra. Performance analysis of MPI collective operations. In *Cluster Computing*, 2005.
31. S. Robert, S. Zertal, and G. Goret. Auto-tuning of io accelerators using black-box optimization. In *Proceedings of the International Conference on High Performance Computing Simulation (HPCS)*, 2019.
32. Sophie Robert. *Auto-tuning of computer systems using bmack-box optimization: an application to the case of I/O accelerators*. PhD thesis, University of UPSaclay, 11 2021.
33. Sophie Robert, Soraya Zertal, and Philippe Couvee. Shaman: A flexible framework for auto-tuning HPC systems. In *Modelling, Analysis, and Simulation of Computer and Telecommunication Systems - 28th International Symposium, MASCOTS 2020, Nice, France, November 17-19, 2020, Revised Selected Papers*, volume 12527 of *Lecture Notes in Computer Science*, pages 147–158, 2020.
34. Sophie Robert, Soraya Zertal, and Philippe 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)*, pages 147–158, 2021.
35. Sophie Robert, Soraya Zertal, Grégory Vaumourin, and Philippe Couvée. A comparative study of black-box optimization heuristics for online tuning of high performance computing i/o accelerators. *Concurrency and Computation: Practice and Experience*, 2021.
36. K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *2008 IEEE International Conference on Cluster Computing*, pages 421–429, 2008.
37. Florian 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*, pages 1826–1835, 2013.
38. Hari Subramoni, Krishna Kandalla, Jérôme Vienne, Sayantan Sur, William Barth, Karen Tomko, Robert Mclay, Karl Schulz, and D.K. 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)*, pages 317–325, 2011.
39. Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. In *International Journal of High Performance Computing Application*, volume 19, pages 49–66, 2005.
40. Bibo Tu, Ming Zou, Jianfeng Zhan, Xiaofang Zhao, and Jianping Fan. Multi-core aware optimization for mpi collectives. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pages 322–325, 2008.
41. Y. Hamadi V. K. Ky, C. D'Ambrosio and L. Liberti. Surrogate-based methods for black-box optimization. *International Transactions in Operational Research*, (24), 2016.
42. S.S. Vadhiyar, G.E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000.
43. S.S. Vadhiyar, G.E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000.
44. 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*, pages 670–677, 2019.
45. Karin Zielinski, D. Peters, and R. Laur. Stopping criteria for single-objective optimization. 2005.