Application-Agnostic Auto-tuning of Open MPI Collectives using Bayesian Optimization

Emmanuel Jeannot
Inria, Labri, U. Bordeaux
Talence, France

Pierre Lemarinier
ATOS
Échirolles, France

Guillaume Mercier
Bordeaux INP, Inria, Labri, U. Bordeaux
Talence, France

Sophie Robert-Hayek
ATOS, U. Lorraine
Échirolles, France

Richard Sartori
ATOS, Inria, Labri, U. Bordeaux
Échirolles, France

***Abstract*—MPI implementations encompass a broad range of parameters that have a significant impact on performance, and these parameters vary based on the specific communication pattern. State-of the art solutions [25], [15] provide a per application tuning of these parameters which require to do the tuning for each application or redo it each time the application changes. Here, we propose an application-agnostic method that leverages Bayesian Optimization, a black-box optimization technique, to discover the optimal parametrization of collective communication. We conducted experiments on two HPC platforms, where we tune three Open MPI parameters for four distinct collective operations and 18 message sizes. The results of our tuning exhibit an average execution-time improvement up to 48.4% compared to the default parametrization, closely aligning with the tuning achieved through exhaustive sampling. Additionally, our approach drastically reduces the tuning time by 95% in contrast to the exhaustive search, achieving a total search time of merely 6 hours instead of the original 134 hours. Furthermore, we apply our methodology to the NAS benchmarks, demonstrating its efficacy and application agnosticity in real-world scenarios.**

***Index Terms*—Message Passing Interface, High Performance Computing, Auto-tuning, Black-box optimization**

I. Introduction

Message Passing Interface (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. One of its key features is MPI collective communication operations, as they provide a standardized interface for performing data movements within groups of processes. Several major implementations of this standard exist and are ubiquitous in HPC centers. These implementations feature a runtime system that is tailored to the underlying architecture to improve communications performance. A usual strategy for better adapting such a runtime system comes with user parametrization, allowing one to specify, for instance, which type of network to use when different ones coexist or to select the collective communication algorithm and its configuration.

As HPC systems and programs grow at scale, communication speed lags behind. In modern production systems, up to 50% of execution time is spent in MPI routines and not on actual computations [13]. We expect communication overhead to become an even larger bottleneck in future exascale machines.

Choosing the optimal set of parameters is a crucial task to improve the performance of HPC applications, as the default parametrization is often suboptimal and strongly depends on the specificities of the communication problem. This is known to be a challenging issue [7] that requires time and expert efforts, as a collective communication depends on the used algorithm and its implementation but also on the application-specific communication pattern, including, for instance, the size of messages or the number of processes involved. For a given collective communication operation, major MPI implementations such as Open MPI [8] or MPICH [1] provide default settings depending on different parameters, mainly the size of the message and the number of MPI processes. However such default parametrization is often suboptimal, as we will show in the experiments section of this paper. Hence, these implementations allow the user to choose among numerous collective algorithms and provide ways to configure each of these algorithms. The main motivation behind our work is that the correct setting of runtime parameters for a given application and platform results in a valuable performance gain when compared to default parameters.

A possible solution to find the optimal parametrization is to perform an exhaustive sampling of the parameter space, but the very large size of this space makes this method too costly to be efficient in a production environment. For instance, the all_to_all collective operation of Open MPI features six algorithms, their thresholds (for small, medium and large message size), a fan-out parameter (for tree or chain algorithms) and two other parameters (min_procs and algorithm_max_request). Moreover, each of these parameters depends on the message size and number of MPI processes involved in the operation. Another solution is to build an analytical model that represents each different communication problem and selects the best parametrization according to this model. However, building such a model is often impractical because of the required insight into the behavior of the system and the collective operations' algorithms.

Furthermore, the ideal configuration varies based on the application and the specific target machine. Recent research has introduced application-specific tuning [25] for individual target machines. However, these approaches necessitate re-tuning whenever changes to the application occur or when a new one runs.

To overcome these shortcomings, this paper proposes to tune each collective operation once and for all (given a target machine) in order to generate a configuration file usable by any application running on the machine. We achieve this autotuning using a black-box optimization which provides a way to find the optimal value of a function without making any hypothesis on its behavior. Black-box optimization has demonstrated some promising results in different optimization fields such as energy consumption [33], I/O accelerators [38] or high-end storage bays [6]. Since it does not require previous knowledge on the relationship between the parametrization of the collectives and the performance of the application, it is easier to implement than an analytical model. The main interest of these methods compared to exhaustive sampling is that, since it relies on guided search heuristics, it explores the parameter space much more efficiently and can converge quickly toward the optimal solution or an approaching one. This introduces a trade-off between time and accuracy of the tuning that leans in favor of the guided search method as the parameter space of the tuning problem is often simple enough to converge very quickly. More precisely we use in this paper Bayesian Optimization [17] as the black-box optimization algorithm to address our problem. We describe this method for tuning an MPI implementation runtime parameters with a focus on collective communication operations. Our target implementation is Open MPI, for which we tune four major collective communication operations through various parameters, such as the message size, the number of nodes and the number of processes. To evaluate our approach, we compare the tuning time of Bayesian Optimization with exhaustive sampling (*i.e.* brute-force) and assess the resulting configurations against the default parametrization's performance in terms of execution time. We show how to apply these tunings to real applications.

The core contribution of this paper represents a significant advancement through the application of Bayesian Optimization. Compared to other existing works, our approach does not need to tune applications individually. Moreover, our method not only drastically reduces the tuning time for four essential MPI collective communication operations, but also ensures that their performance is not compromised relative to their optimal configurations. In summary, the contributions of this paper are as follows:

- An application-agnostic approach that does not necessitate a specific tuning for each distinct application as it is applicable to any application without a priori knowledge.
- Up to an average execution time improvement of 48.4% compared to the default parametrization.
- A low average execution time difference of only 6% when compared to the optimal parametrization.
- A reduction in total tuning time from 134 hours (using exhaustive sampling) to a mere 6 hours, reflecting a 95% gain in efficiency (time and energy).
- A demonstration of the scalability of our solution, proving its adaptability to diverse computing node configurations.

Overall, these results highlight the potential and practicality of our Bayesian Optimization technique to improve MPI performance in diverse computing scenarios. It also shows that application-independent tuning of collective communication operations can achieve close to the optimal solution.

The remainder of this paper is organized as follows: Section II describes the main context of the study and justifies the importance of running MPI collective communication operations with the best possible parameters. Section III presents the related work while Section IV gives a description of the methods used to find the optimal parameters. The validation plan for the evaluation of our method is described in Section V, and the corresponding results are described in Section VI. Future work and conclusion are detailed in Section VII.

II. Context and problematic

The MPI standard defines a set of collective communication operations, which involve the participation of a subset of all processes. Instead of using point-to-point communication, applications can utilize these collective communication operations to transparently leverage performance improvements offered by the MPI implementation. MPI libraries often offer multiple implementations for each collective communication operation that can be configured independently. According to previous studies [16], [20], the optimal configuration depends significantly on the size of the transmitted message, as well as the architecture and topology of the target platform.

A. Open MPI Tuned collectives

The Open MPI implementation is a widely used open-source implementation of the MPI standard, deployed in large-scale HPC centers. It features a modular architecture, where different modules may implement the same set of operations and can be chosen at runtime based on the user's decision.

The Open MPI implementation is also divided into frameworks, including the *coll* one that provides the collective communication operations. The "tuned" module [21] is a network-agnostic implementation that focuses on performance by decomposing a collective communication operation into point-to-point operations tailored for the underlying architecture. This module proposes for each collective communication operation various algorithms implementations such as tree-based or ring-based ones (for instance), and allows the user to tune the algorithms on, for example, the degree of tree-based algorithms, or the message segmentation size.

For the purpose of demonstrating our approach, we have selected four collective communication operations, but the process can trivially be expanded to all the others. The selected collective communication operations are among the most widely used in HPC applications, and cover all communication patterns (i.e, one-to-all, all-to-one and all-to-all):

- **Broadcast**: Broadcast is a collective operation where one process sends the same data to all processes.
- **Gather**: The gather collective operation takes data from several source processes and gathers them onto one single root process.
- **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.
- **Allreduce**: The goal of the allreduce operation is to reduce the values and distribute the results to all processes.

B. Tunable parameters of Open MPI

Open MPI provides a simple interface for tuning its runtime environment [34] with so-called Modular Component Architecture (MCA) parameters. Several hundreds of such parameters are available for tuning but in this paper we focus on the subset of parameters related to the coll_tuned component that are used during a collective communication operation and that can be set dynamically: **algorithm, fan-in/fan-out and segment size** as described in table I.

C. Setting MCA parameters

Each MCA variable can be set individually via parameters of the `mpiexec` command or in an aggregated way through a configuration file. The MCA variable `coll_tuned_use_-dynamic_rules` should be set to 1 for it to work. Listing 1 gives an example of such a configuration file.

```
1   # number of rules for collectives
2   # Id of the collective (allreduce)
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     # size id faninout segsize
1024  1  4  64  # size id faninout segsize
```

Listing 1. Example of an Open MPI configuration file

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 shall apply: if the message size is lower than 1024 bytes, the 7th algorithm shall be used, with a fan-in/out value of 4 and a segment size value of 32, and if the message size is greater than 1024 bytes, the first algorithm will be used, with a fan-in/out value of 4 and a segment size value of 64. This file format allows to set any number of rules and conditions for the applications. To use this file as the Open MPI configuration file, the variables `coll_tuned_dynamic_rules _filename` and `_fileformat` should be set to the path to your configuration file and to 1, respectively.

D. Impact of parameters on performance

Open MPI provides a default parametrization that is independent of the target machine and therefore often suboptimal. Indeed, the importance of selecting the right parameters compared to the default ones when running an MPI application is illustrated by Figure 1. It shows the gain (in percent) of the execution time for each collective benchmark when using the best possible parametrization relative to the default one[1]. We observe a noticeable gain (up to 100%) in execution time for most configurations, illustrating the importance of optimizing the parameters of MPI collective communication operations. This gain is also highly sensitive to the message size, the collective operation used and the number of MPI processes involved, which motivates our approach of relying on autotuning methods.

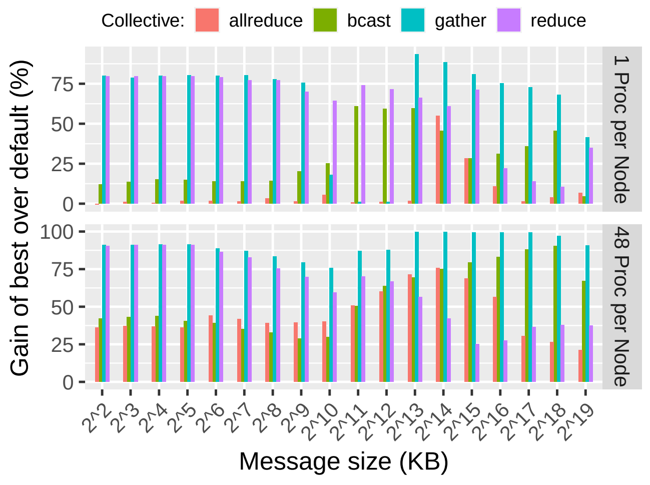


Fig. 1. Execution time gain (in percentage) for the best parametrization over the default one (Pise machine)

E. The limits of brute force

The best parametrization depicted in Figure 1 was found using brute force, *i.e.* running every single possible parametrization. The number of tested parametrizations per collective communication operation, as well as the time required to test them all, are shown in Table II. Since the tuning can last for several days (e.g., an exhaustive tuning of the broadcast operation takes more than four days), brute force is impractical on a production system, as it would require way too many computing resources, user time and energy.

III. Related Work

Auto-tuning using black-box optimization has been successful in finding (near-)optimal configurations of software and hardware systems. For instance, in [10], the authors compare two derivative-free methods to find the optimal configuration of the Hadoop framework, and Bayesian Optimization [17] has been used to find the optimal configuration of the Apache

[1]The method and the benchmarks used to obtain optimal parametrization is further discussed in resp. Sec. II-E and V-A

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

TABLE I
DESCRIPTION OF RUN-TIME PARAMETERS OPTIMIZED IN THIS STUDY AND THEIR DEFAULT VALUES (REPLACE * BY THE NAME OF A COLLECTIVE COMMUNICATION OPERATION)

Collective	Parameter space	Tuning time (min)
Allreduce	1400	505
Bcast	2000	5842
Gather	800	1064
Reduce	1600	639

TABLE II
SIZE OF THE PARAMETER SPACE PER COLLECTIVE AND EXHAUSTIVE SEARCH TUNING TIME IN MINUTES (PISE MACHINE).

Storm computation system [28]. Additionally, a more general tuning framework, BOAT, based on structured Bayesian Optimization, is described in [9]. Within the HPC community, auto-tuning has gained attention for tuning HPC applications in order to improve their portability across architectures [11]. Studies in [40], [30] provide comparisons of several random-based heuristic searches for code auto-tuning, while [4] yields good results with surrogate modeling using boosted regression trees. Furthermore, Bayesian Optimization is beneficial for the energy efficiency in HPC system [33], whilst successful tuning of storage systems was achieved using black-box optimization tuners [5], [14]. Reinforcement learning has been used as an auto-tuner to optimize the performance of the Lustre file system in data center storage systems [14], and genetic algorithms were used to find an optimal parametrization for the layers of the HDF5 library [12]. An extension of this auto-tuner that selects the best parameters according to the I/O pattern is described in [2].

In the context of MPI collective communication operations tuning, prior studies have shown that the best communication algorithm for a collective operation highly depends on the message size [16], [41], [18]. Since no algorithm can fit all sizes, MPI implementations provide a set of tunable parameters to configure functions according to the expected behavior. OTPO [7] is the standard tool used by the Open MPI community for tuning MCA parameters. It employs an exhaustive search method, similar to `mpitune` from Intel MPI, to find the optimal parametrization, but this approach can lead to prohibitive tuning times and limits the number of selected parameters or configurations to tune.

Another approach presented in [20] and [16] uses an offline model to predict the effect of runtime parameters on MPI applications to decide which values to use. However, this method is strongly dependent on the dataset used to train it and on the underlying hardware, which can introduce bias and compromise the correlation between parameters and their effects. In contrast, our proposed approach relies on unsupervised and iterative optimization methods, avoiding the need for training and prior knowledge of the optimization problem [24]. This makes it completely independent of the tuned application and hardware, providing a more efficient and flexible tuning process.

Wilkins et al. [19] proposed an approach (FACT) that focuses on minimizing the amount of data fed to the autotuners and that shows remarkable resilience to the decrease of the number of runs of the application. However, it does not consider other influential parameters such as the segment size nor fan-in/out and the employed performance model (*Random-ForestRegressor*) suffers when the number of dimensions of the parameters space increases [3]. But more importantly, the FACT approach/framework is not applied to any application and thus fails to showcase its benefits in terms of performance improvements. Last, FACT's approach is not fully application-agnostic (as stated in their *Future Work* section): "*Considering training data must be recollected as frequently as every job allocation, FACT-based collective autotuning is only practical for longer-running jobs.*"

The successor to FACT, ACCLAiM [15], introduces an auto-tuning machine learning-based method for optimizing collective communications. This time, the method is effectively applied to real applications. However, ACCLAiM's tuning process takes place at runtime and implies a model training for every job run on the target system. In contrast, our approach is truly application-agnostic and optimizes independently each collective communication operation for the target system once and for all. An application is then able to select transparently the best algorithm when it calls a collective communication routine. The tuning in ACCLAiM is carried out on a per-application basis, necessitating adjustments whenever a new application is introduced or when there are changes in the communication pattern of an existing one. This stands in contrast to our approach, which requires a single tuning process for the entire cluster, applicable universally.

A fundamentally different approach was proposed by Hunold et al. [25] in their OMPICollTune solution. It integrates the probing of collective algorithms directly in the MPI library. While this tuning method has shown interesting results, the exploration of the parameters space is solely guided by randomness. This might lead to a significant slowdown when selecting a very slow algorithm, while our approach uses Bayesian techniques to specifically avoid hitting such algorithms. In addition, the OMPICollTune requires modifications of the Open MPI implementation itself, thus not working as-is with it nor with its derivatives.

IV. Bayesian optimization for system auto-tuning

A. Application-Agnostic Optimization Workflow

To improve the process of enhancing various applications without individually optimizing each one, as seen in prior approaches (e.g., [25]), our proposed method relies on the tuning of collective operation benchmarks and then on the exporting of the tuning obtained via a configuration file. The workflow can be summarized as follows:

1) Select the appropriate OSU benchmark [32] corresponding to a given collective communication operation. OSU benchmarks provide performance output for a specific number of nodes and a range of message sizes.
2) Utilize Bayesian Optimization (as detailed in subsequent sections) to fine-tune the selected benchmark for a particular number of nodes and message size.
3) Incorporate this set of tuned configurations into a configuration file, where each collective communication operation is optimized (as shown in Listing 1).

The output configuration file is produced once for a given target machine and for a set of message sizes and number of nodes. Then, when executing an application, we pass the configuration file to the Open MPI runtime system through its MCA parameters, enabling the override of default settings. This ensures an application-agnostic solution since it provides the necessary flexibility to adapt to any application. Its also does not require any modification to the application code.

B. Problem formalization

Let S be the MPI library to optimize and θ_i its parametrization, which belongs to a discrete subset of the possible parametrizations of the MPI libraries $\Theta = \{\theta_i\}_{i \in \mathbb{N}}$. Let $\mathcal{A}$ be a program measuring the performance of the collective communication operation chosen to be optimized within the library S. Let $\mathcal{E}$ be the execution context for which we want to optimize the benchmark. Let F_s be the performance function associated with the application, the execution context and the MPI library: $F_S : (\mathcal{A}, \mathcal{E}, \Theta) \longrightarrow \mathbb{R}$. The optimization problem that we are trying to solve is:

$$\min_{\theta_i \in \Theta} F_{S,\mathcal{A},\mathcal{E}}(\theta_i)$$

In our situation, $\mathcal{A}$ would be one of the benchmarks described in Section V-A, Θ is the Cartesian product of all possible values of parameters listed in Section II, and $\mathcal{E}$ is the platform used for the experiments, presented in Table III.

C. Black-box optimization for auto-tuning

Black-box optimization refers to optimizing a function with unknown properties that are often costly to evaluate, resulting in a limited number of possible evaluations. These methods are promising for tuning various systems, including computer systems. When applied to computer system tuning, the approach treats the system as a black-box, analyzing the relationship between input and output parameters as described in Figure 2. In this context, the black-box comprises the combination of the collective communication operations to optimize, the used MPI library and the execution context (target hardware). The input parameters are those of the Open MPI library, and the output is an assessment of the benchmark performance, expressed for instance as the execution time.

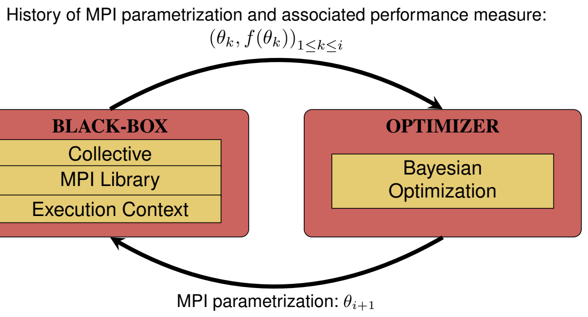

History of MPI parametrization and associated performance measure:
$(\theta_k, f(\theta_k))_{1 \leq k \leq i}$
BLACK-BOX
Collective
MPI Library
Execution Context
OPTIMIZER
Bayesian Optimization
MPI parametrization: θ_{i+1}

Fig. 2. Schematic representation of the optimization loop

The first step of any black-box optimization algorithm is the selection of the initial parameters to start the optimization process. An acceptable initialization starting plan must respect at least two properties [29], [39], [23]: the space constraints property 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. It ensures that if an axis of the parameters space is removed, then no two points would have the same coordinates. This is especially important as we have no insight on the individual effect of each parameter. Latin Hypercube Sampling [29] is a usual choice as an initialization strategy, because of its simplicity and its efficiency [31]. We shall rely only on this method for initialization. The second step of black-box optimization is a feedback step. It consists in iteratively selecting a parametrization, evaluating the black-box function at this point and selecting the next data point to evaluate accordingly by using a higher procedure for searching an optimal solution in a parameter space, called an optimization *heuristic*. Several optimization heuristics are available in the literature, and we chose to focus on one of the most popular and promising, called Bayesian Optimization.

D. General principles of Bayesian Optimization

Besides the initialization plan, Bayesian Optimization requires two inputs to be instantiated: the acquisition function and the probabilistic model used to represent the performance function.

a) The acquisition function: An acquisition function indicates for each configuration its potential performance improvement by being evaluated next, given the input of the probabilistic model. It should offer a trade-off between *exploration* of parameter zones where the model is uncertain (*i.e.* zones with a high variance) and the *exploitation* of already promising well-explored zones (*i.e.* zones where the mean is low). One of the most common acquisition methods is the *Expected Improvement* (EI), which computes the expected improvement from switching from f^*, the best configuration seen so far, *i.e.*:

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

$$EI(\theta) = \mathbb{E}(I(\theta))$$

I represent the relative improvement and EI denotes the expected improvement, computed as the expectancy of the relative improvement. As EI is one of the most popular algorithms and has been proven to be an efficient acquisition function to solve a wide range of problems, we will focus solely on this acquisition function.

b) The probabilistic model: 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 is Gaussian Processes which generate distributions over functions used for Bayesian non-parametric regression. While other methods, such as Parzen trees and Random Forests, have proven to be effective [26], we will focus solely on Gaussian Processes, because of their efficiency in low-dimensional numerical input spaces [26]. A Gaussian Process [36] is fully characterized by a mean function μ at each parametrization, as well as a covariance function Σ between all of the parametrizations of the parameters grid. Mean and variance predictions at parametrization θ_i are obtained as: $\mu(\theta_i) = k_* K^{-1} y$ and $\sigma^2(\theta_i) = \Sigma(\theta_i, \theta_i) - k_*^T K^{-1} k_*$

where 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 Σ has a strong impact on the performance of the model[27], and we opted for the common choice of a radial-basis function kernel (with d the Euclidean distance): $\Sigma(\theta_i, \theta_j) = exp(d(\theta_i, \theta_j))$

c) Stopping criterion: The optimization process runs until either the maximum number of steps is exceeded or a stopping criterion evaluates to true. For our use-case, we chose a maximum number of iterations set to 150 and choose to stop the optimization process if the best execution time over the last 15 iterations is less than 1% better than the current found minimum.

V. Experimental Validation

The goal of the validation plan is to assess the performance of using Bayesian Optimization for tuning a set of collective communication operations benchmarks, comparing the resulting parametrization against the default provided by Open MPI and the one obtained with an exhaustive search in the parameter space, both in terms of execution time and tuning time. Moreover, as the works of [25], [15] do not provide the code we have compared our approach to the optimal case found using brute-force. Anyways, we advocate that comparing against the optimal is more challenging than against the state-of-the-art.

For our experiments, the artifact description can be found here: https://www.labri.fr/perso/ejeannot/artifact.html

A. Selected benchmarks

The tuning of the collective is performed using the OSU Micro-Benchmark [32], which is a suite of MPI benchmarks that implements notably a performance evaluation for every collective communication operation. For each of the tuned collective communication operation and each tested size, we use the corresponding benchmark in the suite.

B. Experimental plan

The evaluation of the performance of Bayesian Optimization is carried out by tuning the four collective communication operations introduced in Section II, for a message size ranging from 2^2 to 2^{19} bytes, with a multiplicative step of 2. We used the clusters described in Table III. We emulate two of the most common types of resource assignments and process placement policies encountered in HPC applications, each time using a multiple node count up to the limit of the cluster (with an additive step of 6).

- *One MPI process per node*: 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. In this case, the MPI process is bound to a specific core within the node.
- *One MPI process per core*: This type of setting is typical of pure, MPI-only applications that rely on the MPI library for all their communications (inter and intra-node alike). To ensure reproducibility, MPI processes are bound to their respective cores in a linear fashion.

This results in a total of 1296 optimization experiments (2 clusters, 4 or 5 node counts, 4 collective operations, 18 message sizes and 2 different placement policies). For each of these experiments, the parameter space is composed of segment size, fan-in/out and collective communication algorithm identifier. The optimal point in this space will then be reported in the configuration file for Open MPI to use.

C. Methodology

To compare the execution time, the default parametrization is run one hundred times to account for possible noise. Exhaustive sampling (*i.e.* brute force) of the parameter space is then performed in order to get the corresponding execution time at each possible parametrization. We select the parametrization with the minimal execution time as the optimal one, which acts as the baseline and is also run one hundred times for noise mitigation.

The tuning of the collective communication operation is performed using Bayesian Optimization, as described in Section IV. The best parametrization is also run one hundred times to account for noise.

D. Hardware platform and tools

All of the tests are run on the `Pise` and `Bora` platforms, described in Table III.

Platform	`Pise`	`Bora`
Node count	32	24
Open MPI version	4.0.4	4.1.5
CPU	2 x AMD Rome 24 cores (AMD EPYC 7402)	2 x Intel Skylake 18 cores (Xeon Gold 6140)
Interconnect	Mellanox ConnectX6 HDR200 (pcie4)	OmniPath 100GBit/s

TABLE III
HARDWARE DESCRIPTION

To perform the tuning, we use the SHAMan framework for auto-tuning of HPC components [37], configured specifically for the Open MPI implementation. We use this framework for both the exhaustive search and the Bayesian Optimization.

VI. RESULTS

A. Execution time Comparison between Bayesian Optimization and Default Parametrization

The execution time gain of using the best parametrization found by the Bayesian Optimization compared to the default one is represented in Figure 3 for Pise. 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.

The time gain brought by Bayesian Optimization varies depending on the tuned collective communication operation, with some where the default parametrization is more adapted than others. The improvement of the default parametrization is strongly dependent on each evaluated parameter (message size, number of processes per node or type of operation) and is difficult to predict. This highlights the importance of tuning each configuration to get the best performance, and the need for an efficient tuning method.

For the Bora case, we also see a huge gain brought by the best parameterization, as shown in Figure 4, with an average gain of 39.3%. For the single process per node case, results are very similar to the Pise with an average, the gain is 45.0% using the best parameterization. We also see how machine dependent is the tuning: the gain of the allreduce is closer to 100% for large message on the Bora machine while the default parameterization was almost optimal on the Pise machine. For one process per core (36 processes per node), we observe that for small message sizes, the gain can be very large for gather but relatively small when dealing with large messages. Nevertheless, in this case, the average gain is 33.6%.

B. Execution Time Comparison between Bayesian Optimization and Brute Force

The median difference in elapsed time, along with the noise measurement, between the best parametrization found by Bayesian Optimization and the optimal parametrization found by exhaustive search is represented in Table IV. When

Collective	# of MPI proc.	ΔT (μs)	Noise (μs)
allreduce	12	0.04	0.43
	576	3.80	1.05
bcast	12	0.26	0.32
	576	0.18	0.32
gather	12	0.01	0.04
	576	0.00	0.02
reduce	12	0.00	0.06
	576	0.00	0.03

TABLE IV
MEDIAN DIFFERENCE IN EXECUTION TIME AND NOISE BETWEEN BEST PARAMETRIZATION FOUND BY BAYESIAN OPTIMIZATION AND OPTIMAL PARAMETRIZATION (PISE MACHINE)

looking at the different collective communication operations and hardware platforms, we find an average distance of 6% and the difference between the two optimal parametrizations to be inferior to the measured noise, for all collective operations except for the allreduce case with 576 MPI processes. While the tuner has not been able to find the optimum in these cases, we find that the difference in performance is negligible for applications running in production and the gain compared to the default parametrization is enough to justify the benefit of Bayesian Optimization.

C. Tuning Time Comparison

The elapsed time required to reach the optimum for each of the collective communication operations and hardware configurations is reported in Table V. With a time gain of more than 85% for each collective operation, we see the difference coming from using guided search heuristics instead of testing every parametrization with exhaustive sampling. The time required to run all the 1296 optimization experiments ranges from a total of 8048 minutes (5 days and 14 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 communication operation and each target platform, that is why the very similar results for the `Bora` platform are not shown.

Coll.	# proc	Brute force	Bayesian Opt.	Gain (%)
Allreduce	12	53	5	91.40
	576	453	47	89.66
Bcast	12	745	24	96.82
	576	5098	134	97.39
Gather	12	24	4	85.42
	576	1041	78	92.56
Reduce	12	88	6	94.01
	576	551	62	88.82

TABLE V
TIME TO SOLUTION FOR EACH HEURISTIC AND EACH COLLECTIVE, ROUNDED UP TO THE NEAREST MINUTE (PISE MACHINE)

The median number of iterations per collective operation is reported in Table VI. For Bayesian Optimization, the number of iterations is stable across each collective operation and each parametrization (mean number of approximately 30), but the size of the tuned message has an impact on the number of iterations, as shown in Fig. 5(a) and Fig. 5(b), respectively, for the `Pise` and `Bora` machines. Moreover, we see that the

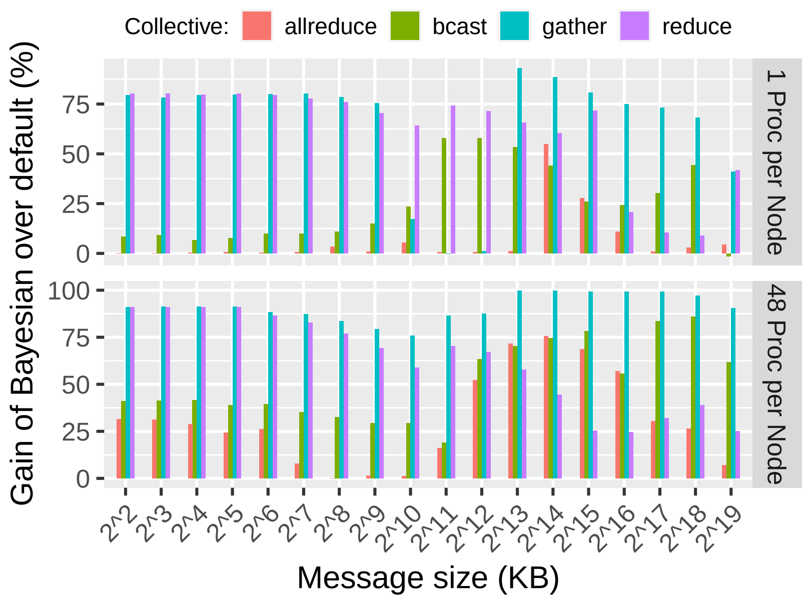


Fig. 3. Execution time gain of using the solution found by Bayesian Optimization compared to the default parametrization (Pise machine)

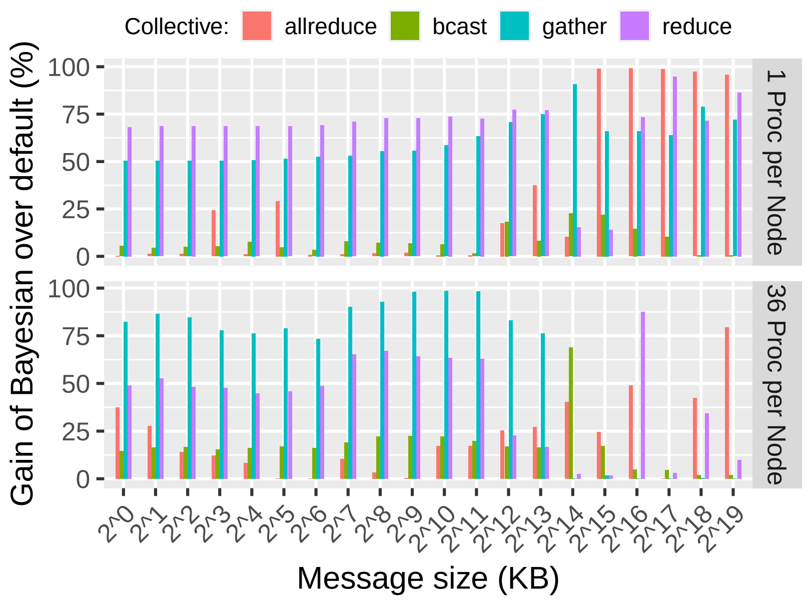


Fig. 4. Execution time gain of using the solution found by Bayesian Optimization compared to the default parametrization (Bora machine)

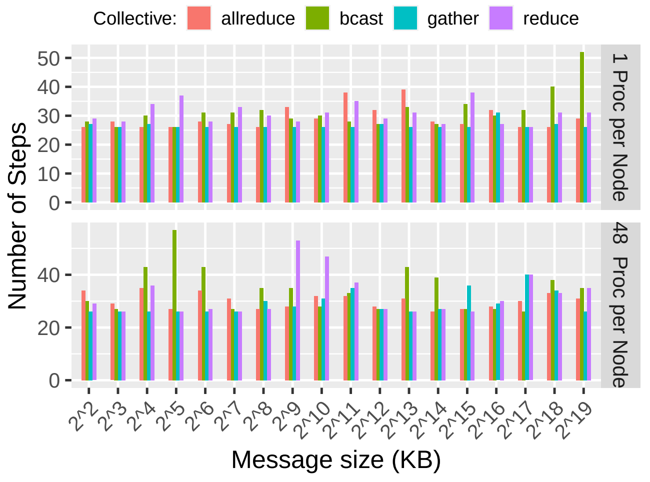
Collective	Exhaustive search	Bayesian Optim.	
		`Pise`	`Bora`
Allreduce	1400	29.0	30.5
Bcast	2000	30.0	31.0
Gather	800	26.0	28.0
Reduce	1600	29.5	31.0

TABLE VI
MEDIAN NUMBER OF ITERATIONS PERFORMED BY BAYESIAN OPTIMIZATION COMPARED TO EXHAUSTIVE SEARCH

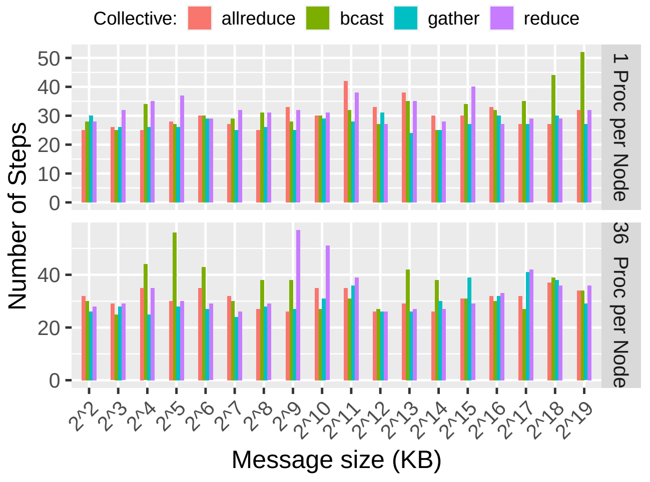
number of steps (e.g. the tuning time) does not depend on the machine but on the number of MPI processes involved. The 150 steps limit mentionned in Section IV has been chosen arbitrarily to set an upper bound on the number of iterations. The table shows that in practice, the process converges before that value is reached.

D. Scalability Study

For the reduce collective operation and a single MPI process per node, the tuning time for different numbers of nodes is shown in Table VII. We see that the number of nodes has a non-linear impact on the convergence of the algorithms. The elapsed time for tuning using Bayesian Optimization



(a) Pise machine



(b) Bora machine

Fig. 5. Number of benchmark runs before reaching stop criterion

is especially stable across the number of nodes because of the early detection of non-promising tuning, but also across the different architectures. The similarity between the two behaviors on very different architectures gives us confidence that our suggested solution is not only application-agnostic but also architecture-agnostic and shall scale well on large-scale production HPC clusters.

# nodes	Brute force	Bayesian Optim.	
		Pise	Bora
6	160.52	4.46	5.71
12	185.32	5.98	7.02
18	209.19	5.81	5.41
24	218.41	6.39	5.04

TABLE VII
TUNING TIME (IN MINUTES) VS. NUMBER OF NODES (FOR THE REDUCE COLLECTIVE OPERATION)

E. Application to the NAS benchmarks

One crucial advantage of the tuning achieved through our method is its independence with regard to the application. Once we discover the optimal tuning for each collective communication operation on the target hardware, we generate an MCA configuration file similar to the one shown in Listing 1. This configuration is then applied to the application by having the `mpiexec` launcher read this file (see Sec. II-C). Our OSU benchmarks only use one collective operation at a time, which is useful to produce a tuning but will not accurately represent regular MPI applications. To see how the tuning applies in a more general context, it was used during the execution of a subset of the NAS applications[22] that is diversified enough to represent other non-benchmarks applications. The results are presented in Table VIII, revealing substantial improvements in execution times. Even though the considered kernels spend only a small fraction of their time in communications, we observe significant gains, particularly in the case of the Conjugate Gradient kernel, which exhibits minimal communication-computation overlap. These findings underscore the profound impact and value of our tuning approach for a wide range of applications.

Class	CG	FT
C	4.5	1.8
D	6.6	1.6
E	5.5	0.4

TABLE VIII
GAIN IN EXECUTION TIME (%) OF NAS KERNELS 32 NODES, 32 PROCESSES PER NODE (BORA MACHINE)

VII. CONCLUSION AND FURTHER WORK

Being able to fine-tune MPI collective communication operations is of paramount importance, as the default parameterization is often sub-optimal on the considered cluster.

In this paper, we propose an application-agnostic and architecture-agnostic workflow to address this problem. It is based on the generation of a configuration file using collective communication operations benchmarks and Bayesian Optimization to find the (near-)optimal parameterization of Open MPI collective communication operations as implemented in the "tuned" component. Our method efficiently balances tuning time and solution accuracy, outperforming brute force methods. As opposed to previous work where the tuning is done on a per-application basis, our method requires the tuning to be done only once and for all per machine architecture.

Tests on four MPI collective communication operations, across different hardware and message sizes, showed that our approach achieved solutions only 6% away from the brute force tested optimum, with a 95% reduction in tuning time (saving machine configuration time and energy). This led up to a significant average execution time improvement of 48.4% compared to the default Open MPI parameterization.

Our tuner scales well and is applicable to large-scale HPC configurations.

Future work includes exploring more tuned parameters and process placement policies, testing on diverse communication problems and platforms to study transferability. We will explore how efficient our proposed method is for tuning the new and more complex "han" component, which implements collective operations by taking into account topological information, but also how it can be adapted to other communication runtime environments.

ACKNOWLEDGEMENTS

Experiments presented in this paper were carried out using the PlaFRIM [35] experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine.

AUTHOR STATEMENT

Emmanuel Jeannot: Writing, Original Draft, Supervision, Visualization ; **Pierre Lemarinier**: Conceptualization, Writing, Original Draft, Supervision ; **Guillaume Mercier** : Conceptualization, Writing, Original Draft, Supervision ; **Sophie Robert-Hayek**: Methodology, Software, Validation, Investigation, Writing, Visualization ; **Richard Sartori**: Methodology, Software, Validation, Investigation, Writing.

REFERENCES

[1] MPICH: a high performance and widely portable implementation of the Message Passing Interface (MPI) standard. https://www.mpich.org/, 2007.

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

[3] Mariana Belgiu and Lucian Drăguţ. Random forest in remote sensing: A review of applications and future directions. *ISPRS Journal of Photogrammetry and Remote Sensing*, 114:24–31, 2016.

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

[5] Z. Cao. *A Practical , Real-Time Auto-Tuning Framework for Storage Systems*. PhD thesis, SU 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 *USENIX*, USENIX ATC '18, pages 893–907, 2018.

[7] M Chaarawi, J M. Squyres, E Gabriel, and S Feki. A tool for optimizing runtime parameters of open mpi. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 210–217, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/, 2005.

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

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

[11] Balaprakash et al. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, 2018.

[12] Behzad et al. Taming parallel i/o complexity with auto-tuning. In *SuperComputing*, SC '13, pages 68:1–68:12, 2013.

[13] Chunduri et al. Characterization of mpi usage on a production supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 386–400, 2018.

[14] Li et al. Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In *SuperComputing*, 2017.

[15] M. Wilkins et al. ACCLAiM: Advancing the practicality of MPI collective communication autotuning using machine learning. In *IEEE Cluster 2022*, pages 161–171, 2022.

[16] Pješivac-Grbović et al. Performance analysis of mpi collective operations. In *Clust. Comput.*, volume 2005, 01 2005.

[17] Shahriari et al. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[18] Tu et al. Multi-core aware optimization for mpi collectives. In *ICCC*, pages 322–325, 09 2008.

[19] Wilkins et al. A fact-based approach: Making machine learning collective autotuning feasible on exascale systems. In *2021 Workshop on Exascale MPI (ExaMPI)*, pages 36–45, 2021.

[20] Zheng et al. Auto-tuning mpi collective operations on large-scale parallel systems. In *HPCC/SmartCity/DSS*, pages 670–677, 2019.

[21] G. Fagg, G. Bosilca, J. Pješivac-Grbović, T. Angskun, and J. Dongarra. Tuned: An open mpi collective communications component. In Péter Kacsuk, Thomas Fahringer, and Zsolt Németh, editors, *Distributed and Parallel Systems*, pages 65–72, Boston, MA, 2007. Springer US.

[22] Ahmad Faraj and Xin Yuan. Communication characteristics in the nas parallel benchmarks. In *IASTED PDCS*, pages 724–729. Citeseer, 2002.

[23] R. Gramacy. *Surrogates: Gaussian Process Modeling, Design and Optimization for the Applied Sciences*. Chapman Hall, 2020.

[24] 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)*, pages 259–269, 2020.

[25] S Hunold and S Steiner. Ompicolltune: Autotuning mpi collectives by incremental online learning. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 123–128, 2022.

[26] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[27] F. Hutter, L. Kotthoff, and J. Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Springer, Cham, 2019.

[28] P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. *CoRR*, 2016.

[29] A Keane and A Sóbester. *Engineering Design via Surrogate Modelling*, chapter 1, pages 1–31. John Wiley & Sons, Ltd, 2008.

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

[31] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[32] OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/, 2011.

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

[34] S. Pellegrini, W. Jie, T. Fahringher, and H. Moritsch. Optimizing mpi runtime parameter settings by using machine learning. In *16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009, Proceedings*, 09 2009.

[35] PlaFRIM. https://www.plafrim.fr/, 2015.

[36] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[37] S. Robert, S. Zertal, and P. Couvee. Shaman: A flexible framework for auto-tuning HPC systems. In *MASCOTS 2020, Nice, France*, 2020.

[38] S. Robert, S. Zertal, G. Vaumourin, and P. 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.

[39] J. Sacks, W. Welch, T. Mitchell, and . Wynn. Design and Analysis of Computer Experiments. *Statistical Science*, 4(4):409 – 423, 1989.

[40] 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.

[41] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. 19(1), 2005.

Artifact Description

The experiments utilize various tools:

- The OSU benchmarks are accessible here: http://mvapich.cse.ohio-state.edu/benchmarks
- Exhaustive search and Bayesian optimizations are performed using SHAMan: https://github.com/bds-ailab/shaman
- The NAS benchmarks can be found here: https://www.nas.nasa.gov/software/npb.html

To use SHAMan with the NAS, the scripts can be downloaded from: https://figshare.com/s/b90077bbe3a2c1f55764. In this tarball, the `HOWTO.md` file provides instructions on how to reproduce the experiments (for the NAS or for the OSU benchmarks). All the SHAMan documentation can be found here https://shaman-app.readthedocs.io/en/latest/.

Guide for Reproducing the Experiments

SHAMAN

- Shaman Installation
  1. Clone the GitHub repository
  2. Refer to the guide for detailed installation instructions
  3. It is recommended to create a Python virtual environment. Starting the worker is not necessary since the web interface is optional.
  4. Create a *shaman.env* file defining the variables SHAMAN_API_HOST and SHAMAN_API_PORT, related to the Docker present on the cluster.
- Creating an Experiment
  1. Run `source /venv/bin/activate`, then `source shaman.env`
  2. Build the plugin for parsing results and place it in */shaman_project/bb_wrapper/tunable_component/plugins*. This folder already contains a plugin for parsing an OSU output. The file *parse_nas_output.py* allows parsing a NAS output.
  3. Register the component to be tuned. The file *nas_cg_component.yaml* describes the component used to tune the NAS CG. Registration is done via `shaman-install nas_cg_component.yaml`.
  4. Describe the experiment in a YAML file. The file *cg_experiment.yaml* contains the experiment description for tuning CG.
  5. Write an sbatch file describing the iteration launch (*default.sbatch*).
  6. Run `shaman-optimize --component-name NAS_CG --nbr-iteration 150 --sbatch-file default.sbatch --experiment-name NAS_CG_DEFAULT --configuration-file cg_experiment.yaml --slurm-dir /where/to/store/slurm/outputs --result-file cg_experiment.out`

MCA_EXTRACTOR

- Installation
  1. Install Open MPI
  2. Export the path to MPI binaries in PATH.
  3. Install Python 3+
- Usage
  1. Launch the command `python3 -i mca_extractor.py`
  2. The list *mca_param_list* then contains all mca parameters listed by the command `ompi_info --all --parsable`
  3. The list *mca_param_tunable_list* contains the subset of tunable parameters from *mca_param_list*
  4. MCA tunable parameters have an iterator among their attributes. It iterates over the acceptable values for this parameter, which can be obtained via the command `list(mca_param_tunable_list[0])` (example for the first tunable mca parameter).