Predicting MPI Collective Communication Performance Using Machine Learning

Sascha Hunold*, Abhinav Bhatele†, George Bosilca‡, Peter Knees*

*Faculty of Informatics, TU Wien, Vienna, Austria
†Department of Computer Science, University of Maryland, College Park, MD, USA
‡Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
E-mail: *hunold@par.tuwien.ac.at, †bhatele@cs.umd.edu, *peter.knees@tuwien.ac.at

***Abstract*—The Message Passing Interface (MPI) defines the semantics of data communication operations, while the implementing libraries provide several parameterized algorithms for each operation. Each algorithm of an MPI collective operation may work best on a particular system and may be dependent on the specific communication problem. Internally, MPI libraries employ heuristics to select the best algorithm for a given communication problem when being called by an MPI application. The majority of MPI libraries allow users to override the default algorithm selection, enabling the tuning of this selection process. The problem then becomes how to select the best possible algorithm for a specific case automatically.**

In this paper, we address the algorithm selection problem for MPI collective communication operations. To solve this problem, we propose an auto-tuning framework for collective MPI operations based on machine-learning techniques. First, we execute a set of benchmarks of an MPI library and its entire set of collective algorithms. Second, for each algorithm, we fit a performance model by applying regression learners. Last, we use the regression models to predict the best possible (fastest) algorithm for an unseen communication problem. We evaluate our approach for different MPI libraries and several parallel machines. The experimental results show that our approach outperforms the standard algorithm selection heuristics, which are hard-coded into the MPI libraries, by a significant margin.

***Index Terms*—Message Passing Interface, Performance Prediction, Auto-tuning, Machine Learning, GAM, XGBoost, KNN**

I. Introduction

Taking in account the ongoing trends, high performance computing (HPC) applications are and will, for the foreseeable future, be built on top of the Message Passing Interface (MPI). MPI defines the syntax and semantics for various types of communication operations between processes in a distributed application. An important part of the MPI standard are the so-called collective operations, i.e., communication applicable between a group of processes. An example of a collective operation is the broadcast operation (`MPI_Bcast`), where one process sends a specific data item to all other participating processes. There are several open-source libraries implementing the latest MPI standard, such as Open MPI [1], MVAPICH [2], and MPICH [3], as well as vendor provided libraries, usually derived from one of the open-source libraries. It is well known that there is no single best algorithm for a specific MPI collective operation such as the broadcast. In fact, the number of processes, the number of compute nodes, and the message size are all decisive factors to select an efficient algorithm. For example, for small message sizes, the number of point-to-point communication steps (latencies) has to be minimized to minimize the overall running time, whereas for large message sizes, the throughput needs to be maximized [4].

Internally, all MPI libraries have a set of algorithms implemented for a particular MPI collective operation [5]. When an MPI collective is called, a heuristic of the library will select the probably best possible algorithm. The selection process decision is made by taking into account the message size and the number of processes for which the collective was called, the process placement and bindings, as well as characteristics of the underlying architectures, such as processor features, network infrastructure and topology. Yet, the decision of which algorithm to choose strongly depends on the actual machine. Therefore, several attempts have been made to automate the process of selecting the best possible algorithm for a collective operation [6], [7], [8]. The existing approaches have several shortcomings, such as limited accuracy or long training time. Hence, the decision logic of which algorithm to select is most often hard-coded into the MPI libraries.

Previously, we had proposed a method to select algorithms for MPI collectives using machine-learning techniques [9]. In this paper, we build upon our previous work and propose a novel method to solve the algorithm selection problem for MPI collectives. Although the new approach has a similar overall prediction scheme, the mechanics and the techniques are completely different. In [9], we built our prediction model on relative speed-up values, whereas now we directly predict running times, which has tremendous advantages for an improved model error. Our initial tests with Random Forest (RF) models worked reasonably well in [9], but for a larger number of datasets, it turned out that other regression learners produce better predictions (e.g., XGBoost, KNN, or GAM). Another significant difference is that we are now able to consider algorithmic parameters in the prediction. For example, for large messages, it is often advantageous to segment messages in order to improve the throughput, and these segment sizes are now incorporated into our models.

In summary, this paper makes the following contributions:

1) We show how to collect training data for MPI collectives in a completely predictable manner. Having an upper bound

978-1-7281-6677-3/20/$31.00 ©2020 IEEE
DOI 10.1109/CLUSTER49012.2020.00036

on the duration of the experiments is extremely important when benchmarking MPI collectives on unknown systems, to avoid spending the compute-hour budget on large supercomputers.

2) We propose an algorithm selection approach for MPI collective operations. Our selection method internally uses a set of regression models. Since our approach should be as robust and as practically applicable as possible, we show that our general approach works with different supervised learning techniques, such as XGBoost, KNN, and GAM. It works out-of-the-box without a lengthy search for the best hyper-parameters.
3) We evaluate our algorithm selection strategy by comparing the performance of the predicted algorithm to the performance of the default algorithm, which was selected by the hard-coded decision logic. The experimental results, which have been obtained on different parallel machines and for different MPI libraries, show a strong evidence that our approach outperforms the standard selection strategies by a large extent.

The paper is structured as follows. We introduce the algorithm selection problem for MPI collectives and our notation in Section II. Then, we outline our approach to solve this problem by fitting a series of regression models in Section III. The experimental analysis is divided into two parts: the setup is detailed in Section IV, and the discussion of the results can be found in Section V. We summarize related works in Section VI and discuss the main results in Section VII.

II. The Algorithm Selection Problem for MPI Collectives

Let us start by introducing the main terminology used in this paper. The main goal of this work is to find the best algorithm for a specific instance, according to a set of optimization criterion. Instances in the MPI context are communication problems of the following type: What is the fastest algorithm on machine M to perform an `MPI_Bcast` operation of m Bytes over p processes? An instance I is characterized by the actual collective call F, a message size m, and the number of processes p. MPI libraries contain a set of algorithms A_F for "solving instance" I for a collective operation F. For `MPI_Bcast`, MPI libraries typically provide implementations of the binary-tree, the binomial-tree, or the chain algorithm. Each of these algorithms for `MPI_Bcast` may have its own set of parameters $p_1, \ldots, p_k$ that influence its performance, such as the segment size or the tree fanout.

Given an instance I of a collective communication problem, our goal is to select the algorithm $A_{F,j}$ from the set A_F that completes this communication operation with the shortest execution time overall (which means that for rooted collective it is not enough to simply delegate the heavy work of the collective away from the root process), where j denotes one of the algorithm implementations available for collective F, i.e., $0 \leq j < |A_F|$. It is important that $A_{F,j}$ may only be the fastest algorithm when all its parameters $p_1, \ldots, p_{k_j}$ have been set correctly.

In the literature, these problems are known as the *algorithm selection* [10] and the *algorithm configuration* problem [11]. In an algorithm configuration problem, the goal is to find a well-performing allocation of an algorithm's parameters, such that some performance metric is optimized (e.g., minimizing the running time). The algorithm configuration problem is also commonly known as parameter tuning, and finding the best block size of linear algebra routines on GPUs is one example of this problem type. Note that the configuration problem focuses on one specific algorithm only. In contrast, in an algorithm selection problem, we want to pick the best algorithm from a set of algorithms for a specific instance of the problem.

Now, let us take a closer look at tuning in the context of MPI collectives. As mentioned before, all MPI libraries already use some sort of decision logic to select the fastest implementing algorithm and its parameters. The logic in Open MPI, for example, is based on the work of Pjesivac-Grbovic et al. [8], in which decision trees were built upon a set of benchmark instances and these decision trees were later translated into C code. In addition to the internal selection process, most MPI libraries (e.g., Open MPI and Intel MPI) allow a user to select the actual implementing algorithm for an instance I and collective F depending on the message size and the number of processes. The problem is that selecting the best algorithm is highly dependent on the machine and its architectural characteristics (e.g., the network).

Basic Tuning Options: Since the best algorithm depends on the architecture of machines and their software stack, libraries provide tools to override the default decision logic. Intel MPI, for example, offers tools to auto-tune the algorithm selection decisions for MPI collectives. There are two main problems with these approaches: First, these tuning runs effectively perform an exhaustive search over all algorithms and message sizes for a *pre-selected* number of processes, which is often given as a tuple $p = n \times N$, where n denotes the number of compute nodes and N the number of processes per compute node (often called *ppn*). This exhaustive search is an expensive process, and its *running time* is also *unpredictable*, since the tuning tools rely on benchmarking tools like the OSU Micro-Benchmarks [12] or the Intel MPI Benchmarks [13]. These benchmarking suites repeat the measurements for a pre-defined number of times. This can become very costly in a tuning run, as some algorithms, such as the linear alltoall, may take a very long time to complete, especially with larger number of processes.

Second, when using the tuning results (e.g., in Intel MPI), only the best seen algorithm for all messages sizes and process counts for which the benchmarking was conducted will be returned. Hence, if the tuning run was made on 32×32 processes (32 compute nodes, 32 processes per compute node), it will override the internal algorithm selection strategy for this process count. But if we run the MPI program on 34×32 processes, the default decision logic will be used, as the benchmarking has not been done for this process count. This is also part the problem that we are trying to solve in this

paper. Our hypothesis is that if we know an efficient algorithm for an collective communication problem with instance I_i, the same algorithm will most likely also be efficient for another instance I_k, if I_i and I_k are relatively similar.

Problem Statement: In the present paper, we propose a solution to the algorithm selection problem for MPI collective operations, for cases, where the number of processes per node N is the same on all compute nodes. This is the typical default setting for most batch schedulers like SLURM. While the approach detailed in this paper is generic and could be applied to all collective communications, we focus on blocking MPI collectives, such as `MPI_Bcast`, `MPI_Alltoall`, `MPI_Allreduce`, which are currently the most frequently used collectives according to the study by Chunduri et al. [14]. In particular, we do not consider collectives of the *v*- or *w*-type, where the buffer sizes may vary between processes, as for these collective most often only a few algorithms are available (often one or two).

Figure 1 depicts the algorithm selection problem that we consider. Our approach starts with obtaining a training dataset. To that end, we benchmark the internal algorithms of MPI blocking collectives for different messages sizes, numbers of nodes, and numbers of processes per node. For each case and algorithm, we obtain a performance value, i.e., measure the running time of that algorithm. We use this labeled dataset, our training dataset, to instantiate an algorithm selection strategy. In this paper, we propose one such selection strategy, but there are also other possible strategies (e.g., portfolio builders like AutoFolio [15]). Once the model has been fitted, we apply it to unseen (unknown) instances (e.g., a different number of processes). The overall goal is that the running time of the predicted algorithm (the algorithm identifiers) is as close as possible to the best possible running time of any algorithm.

Our entire approach is an offline strategy, as the benchmarking and the model building have to be done separately before they can be applied. We can apply our model before an MPI application is about to be executed. Once we know how many compute nodes and processes per node have been requested, we query the model for a set of message sizes (10–15 message sizes is enough) and create a configuration file for the different MPI collectives, which can be loaded when starting the application. Hence, the prediction time for a specific instance has not the highest priority. If predictions can be done in the order of seconds, the approach will work seamlessly with SLURM. However, when targeting online approaches, the prediction time needs to be in the microsecond range, as the overhead of MPI collectives would be too large otherwise.

III. Approach to Solve MPI Algorithm Selection Problem

Before devising a strategy for selecting the best algorithm and its parameters for a given MPI collective communication problem, we summarize the essential requirements that such an algorithm selection strategy for MPI collectives must fulfill.

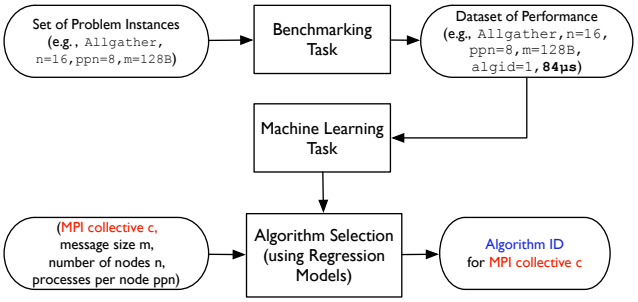

Fig. 1: Overall framework to solve the algorithm selection problem for MPI collectives.

A. Essential Requirements

Practical Applicability of Training Data Selection: A standard procedure for obtaining a training dataset would be randomization. In our context, it would mean that for a given MPI collective, say `MPI_Bcast`, we would measure the performance for a random set of input features, such as message size, number of nodes, or processes per node. This strategy might work for shared-memory systems, but certainly not for supercomputers and compute clusters. On such systems, a batch scheduler will assign our benchmarking job to a subset of machines, whose size is specified as a job parameter. Therefore, our training dataset will comprise the most commonly used input features on a machine, e.g., we train with 16 or 32 compute nodes, as these numbers are commonly used. Moreover, once we have obtained a compute node allocation from the scheduler, we will run as many training runs as possible within a given time frame.

Predictable Training Time: Another requirement is a predictable training time. As mentioned before, literally all MPI benchmark suites measure a collective for a certain number of times (cf. [16]). Yet, the tuning expert wants to precisely define how much time is spent on the training task. For that reason, we need to employ a measurement scheme which allows for setting a benchmarking timer. If the timer is up, the benchmarking run will stop.

Avoid Bias in Training Data: In our previous work [9] on the algorithm selection problem for MPI collectives, we used regression models (random forests) to predict the relative improvement of algorithm $A_{F,j}$ ($j > 1$) with respect to the default strategy $A_{F,0}$. We found two disadvantages with this approach: First, $A_{F,0}$ is not an actual algorithm but a strategy, and thus, the actual algorithm behind $A_{F,0}$ changes depending on the feature vector. It may be that two selected algorithms for $A_{F,0}$ perform not equally well, e.g., one algorithm performs very well for small messages sizes, but the selected one for large messages sizes is not performing well. Thus, the ratio between some algorithm $A_{F,j}, j > 0$ and the strategy (algorithm 0 or $A_{F,0}$) may behave irregularly, which complicates the learning process. The second downside is that the ratios are in the range

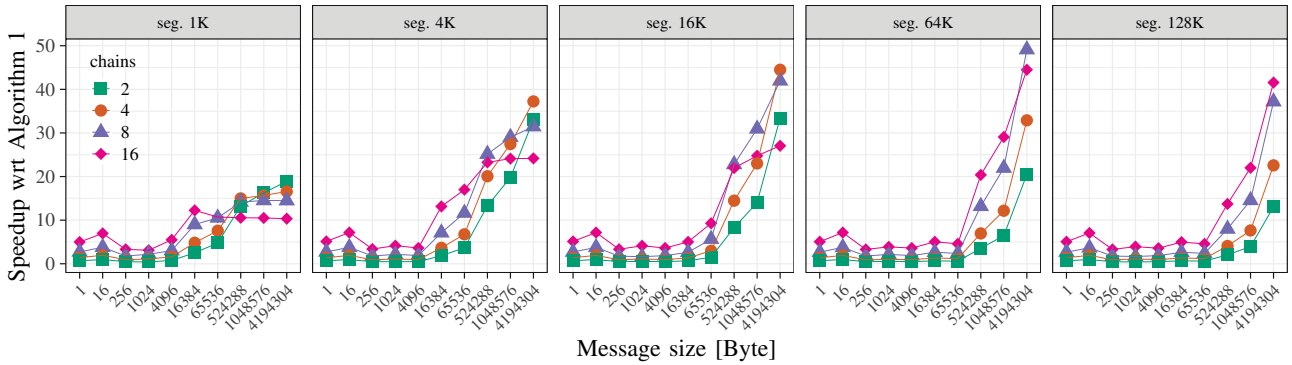

Fig. 2: Speed-up of various algorithmic configurations of alg. 2 (chain algorithm) with respect to alg. 1 (linear) of `MPI_Bcast`; "seg." denotes the segment size while "chains" denotes the number of chains; 32×32 processes, Open MPI 4.0.2, *Hydra*.

$(0, \infty)$, where improvements are ratios with values < 1. This introduces some bias into the learning processes, as several methods try to split the space equally.

Another possible attempt for solving the problem would be to directly predict the algorithm ID for a given feature vector. Each feature in the training data could be labeled with the ID of the best algorithm for that case. The problem is that—in many cases in MPI—a small number of algorithms will perform best on most instances. In these scenarios, the number of different labels in the training datasets would be very heterogeneous, and thus, the final prediction models would be biased towards the heavily used algorithms.

Achieving Robustness and Applicability: A very important, yet often overseen requirement is that our approach should produce good results regardless of the actual machine learning method used. The goal is that by applying a standard regression method, we can get improvements out-of-the-box. In particular, we do not set out to perform any hyper-parameter tuning. Of course, for the final model on a given machine, tuning the hyper-parameters of a specific method (e.g., XGBoost) can improve the model's accuracy. However, in this proof-of-concept, we do not want to rely on hyper-parameter tuning, as the risk of drawing wrong conclusions to due over-fitting the data would be too high.

B. Algorithm Selection and Regression Approach

For a better comprehension of our final algorithm selection strategy, Figure 3 shows an illustration of our method. The basic idea of our approach is to create a regression model for every algorithm $A_{F,j}, j > 0$ that is available for a collective operation F. The goal is to obtain a regression model for each algorithm, which predicts its running time, and then, we select the algorithm that minimizes the running time for an unseen feature vector.

A problem that still needs to be solved is the algorithm configuration problem. As said above, some algorithmic variants possess parameters that effect their performance, e.g., the segment size. In our approach, we combine the ID j of algorithm

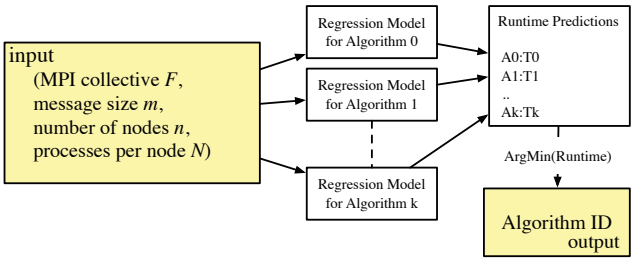

Fig. 3: General algorithm selection strategy for MPI collectives. The running time of each algorithm $(1 \dots k)$ is predicted for a given instance (F, m, n, N). The algorithm with the smallest predicted runtime is selected.

$A_{F,j}$ and a certain allocation of its parameters $p_{1,j}, \dots, p_{k,j}$ to form a unique algorithm identifier $u_{j,l}, 1 \leq l \leq q$, where we assume that the number of different parameter allocations is q. For example, if we consider three segment sizes s_1, s_2, and s_3 for algorithm $A_{F,j}$, the combination of $p_{1,j} \in \{s_1, s_2, s_3\}$ and j, i.e., (j, s_1), (j, s_2), and (j, s_3), would be mapped to a unique identifier $u_{j,l}$. By using this approach, we merge the algorithm configuration and the algorithm selection problem. The limitation is that we need to define the possible values of the algorithms' parameters beforehand. However, in practice, this is rather straight-forward. For example, segmentation is only useful for larger messages, and the segment sizes should not be too small. A reasonable set of segment sizes (e.g., 1K, 4K, 16K, or 64K) is small enough to apply this approach. It is important to consider and to model the different algorithmic parameters. Figure 2 shows the importance of modeling these parameters. Here, we plot the performance ratio of algorithm 2 with respect to algorithm 1 for `MPI_Bcast`. Algorithm 1 in Open MPI is the basic linear algorithm, where the root process sends a message to one process at a time and which has no further parameters. Algorithm 2, the chain algorithm, has two parameters: the segment size and the number of chains. The figure shows that the right choice of these parameters has a significant performance impact, especially for large message

TABLE I: Hardware overview.

Machine Name	n	Max ppn	Processor	Interconnect	MPI library
Hydra	36	32	Intel Xeon Gold 6130, 2.1 GHz Dual socket	Intel OmniPath Dual-rail, dual-switch	Open MPI 4.0.2 Intel MPI 2019
Jupiter	35	16	AMD Opteron 6134	Mellanox InfiniBand (QDR)	Open MPI 4.0.2
SuperMUC-NG	6336	48	Intel Skylake Platinum 8174	Intel OmniPath	Open MPI 4.0.2

sizes. For large ($4\,\mathrm{MB}$) messages, the speed-up of algorithm 2 is between 10 to 50, depending on the values of the parameters.

Next, all algorithms with their respective IDs $u_{j,l}$ (all algorithmic configurations) are benchmarked. For each algorithm, we fit a regression model to predict the running time of a problem instance. To apply our selection approach, we query each regression model and predict the running time of each algorithmic configuration $u_{j,l}$, and the configuration that leads to the shortest running time prediction is finally selected. That means that the resulting algorithm ID not only encodes the selected algorithm $A_{F,j}$ but also the parameter configuration.

C. Regression Model for Running Times

We now explain how to fit a regression model to predict the running time of MPI collectives. As stressed before, our main motivation was to get a working tuning framework that can be applied to any MPI library. Thus, we intentionally omit a rigorous hyper-parameter tuning of the various regression models.

Nonetheless, we tried out several methods from the toolbox for supervised learning, such as Random Forests, Neural Networks, or Linear Regressions. However, they all showed several weaknesses, which is why we finally settled for three other regression methods, which are: XGBoost, K-nearest Neighbor, and generalized additive models (GAM). While testing the prediction quality on our datasets, we noticed that all three performed reasonably well and could all be used inside a practical framework. For the sake of completeness, we opted to show results with all three methods.

Before moving forward with the description of our approach, we describe the selected regression methods. With the exception of XGBoost, a more extended description of these regression methods can be found in Hastie et al. [17].

XGBoost: The XGBoost library (eXtreme Gradient Boosting) [18] uses an ensemble method that combines several weaker classifiers to create a better classifier. The essential method used in XGBoost is a gradient boosting decision tree algorithm.

Generalized additive models (GAM): Several processes, such as the MPI algorithm selection problem, are non-linear and therefore linear regression models fail to provide the necessary prediction accuracy. The GAM method [19] performs a regression on each dependent variable using a scatterplot smoother (e.g., a spline or kernel smoother). These individual functions model possible nonlinearities in the response variable and are then combined (added) into a prediction model.

K-nearest Neighbor (KNN): The KNN algorithm computes distances between points (samples) in the training and test datasets [20]. A common metric is the Euclidean distance. For regression tasks, the KNN algorithm determines the k closest points to an unknown point in the feature space. To determine the final output, the target values of all k closest neighbors are combined (e.g., the mean value is computed).

IV. Experimental Setup

The following section depicts the technical details of our framework. Since we logically perform two different steps, 1) the benchmarking to obtain the datasets, and 2) the building and evaluation of the model, we refer to these steps in the following as the *benchmark step* and the *tuning step*, respectively.

A. Machines

We perform experiments on three different parallel machines, which are called *Hydra*, *Jupiter*, and *SuperMUC-NG*. An overview of their basic properties is shown in Table I. The machines *Hydra* and *Jupiter* are smaller cluster installations at the Vienna University of Technology. *SuperMUC-NG* is in the top ten of the TOP500 list and is located at LRZ Munich. The most obvious differences are the number of cores per compute node and the interconnect. For example, *Hydra* has a dual-rail Intel OmniPath while *Jupiter* has an older single-rail Infiniband interconnect. While being of similar size, *Hydra* has about twice as much bandwidth as *Jupiter* but also twice as many cores. Additionally, the compute nodes vary significantly in the number of cores per compute nodes (from 16 to 48), which is helpful to examine how sensitive the algorithm selection strategy is to the number of cores.

B. Software

In the benchmarking step, we rely on the ReproMPI benchmark [16]. There are two main features that sets it aside from other MPI benchmarking suites. First, it allows for measuring collectives for a predefined benchmarking time. Second, it supports accurate clock synchronization schemes and measuring collectives using a time-window process synchronization [21].

We examine two different MPI libraries: Open MPI and Intel MPI. We use the same version of Open MPI (4.0.2) on all three machines, to avoid drawing wrong conclusions, as the library version would be an additional experimental factor. Our techniques are also applicable to other MPI libraries like MPICH and potentially also to MVAPICH, although MVAPICH uses a slightly different concept for the algorithm selection, where the algorithm for small, medium, or large messages can be altered.

TABLE II: Overview of datasets.

Dataset	MPI routine	MPI	Version	Machine	#algorithms	#nodes	#ppn	#msg. sizes	#samples
d1	`MPI_Bcast`	Open MPI	4.0.2	*Hydra*	9	11	10	10	255200
d2	`MPI_Allreduce`	Open MPI	4.0.2	*Hydra*	7	11	10	10	39600
d3	`MPI_Bcast`	Open MPI	4.0.2	*Jupiter*	9	10	7	10	162400
d4	`MPI_Allreduce`	Open MPI	4.0.2	*Jupiter*	7	10	7	10	25200
d5	`MPI_Allreduce`	Intel MPI	2019	*Hydra*	16	11	10	10	70400
d6	`MPI_Alltoall`	Intel MPI	2019	*Hydra*	5	11	10	8	17600
d7	`MPI_Bcast`	Intel MPI	2019	*Hydra*	12	11	10	10	52800
d8	`MPI_Bcast`	Open MPI	4.0.2	*SuperMUC-NG*	9	5	5	8	23184

In the tuning step, we use the following R packages: `xgboost 1.0.0.2` for XGBoost, `mcgv 1.8` for GAM, and `caret 6.0` for KNN. As emphasized, we do not perform an extensive tuning of the hyper-parameters. Therefore, we use the default $K = 5$ for the KNN. We use scaled inputs for KNN, although we found that our regression models worked slightly better with unscaled inputs. However, that was mostly coincidence, since the message size turned out to be the most important factor in many cases, whose values also happened to be of largest magnitude. Nonetheless, for the sake of a general applicability, we scaled the inputs when using KNN. Since a regression based on linear models, as expected, did not work in XGBoost, we use the `Tweedie` regression (the `Gamma` regression also worked well). We train the model with XGBoost for 200 rounds. For the generalized additive models of `mcgv`, we choose the `Gamma` family for positive, real-valued data and the *log* link function. For more information on these parameters, see Hastie et al. [17].

C. Datasets

In order to evaluate our approach, we measured the performance (running time) of different MPI collective operations for a large number of cases. These datasets are summarized in Table II. Let us take a look at dataset `d1` as an example. This dataset only contains performance measurements of `MPI_Bcast` on *Hydra* using Open MPI 4.0.2, which implements 9 different algorithms to execute the broadcast. We recorded measurements on various numbers of compute nodes n, i.e., $4, 7, 8, 13, 16, 19, 24, 27, 32, 35$, and 36 (hence, #nodes=11). Similarly, we varied the number of processes per node, i.e., `d1` contains values for $N \in \{1, 4, 8, 10, 16, 17, 20, 24, 28, 32\}$ (#ppn=10). We used the following message sizes (in Bytes) for the fixed-sized buffer collectives (Allreduce, Bcast) on all machines: $1, 16, 256, 1024, 4096, 16\,384, 65\,536, 524\,288, 1\,048\,576$, and $4\,194\,304$. Due to space limitations, we cannot provide the full details of the other datasets. The last column "#samples" contains the number of distinct measurements in the dataset. Note that the number in this column is larger than the cross product of #algid × #nodes × #ppn × #msize. The reason is that we also take into account algorithmic parameters such as the segment size or the fanout in the dataset. A specific algorithm $A_{F,j}, j > 0$ is benchmarked on all process configurations (#nnodes × #ppn), all message sizes, but also for all combinations of realistic algorithmic parameters. For example, we tested `MPI_Bcast` in `d1`, with the following segment sizes in KB: $1, 4, 16, 64$, and 128, if segmentation is supported by an algorithm.

TABLE III: Training and test datasets by machine and number of compute nodes (n).

Machine	Full training dataset (n)	Small training dataset (n)	Test dataset (n)
Hydra	$4, 8, 16, 20, 24, 32, 36$	$4, 16, 36$	$7, 13, 19, 27, 35$
Jupiter	$4, 8, 16, 20, 24, 32$	$4, 16, 32$	$7, 13, 19, 27$
SuperMUC-NG	$20, 32, 48$	$20, 32, 48$	$27, 35$

V. Experimental Results

Before we examine the results for each machine separately, we would like to comment on the overall evaluation strategy. In a typical machine learning setting, the prediction error of regression models would be analyzed by metrics like the mean absolute error (MAE) or the root mean squared error (RMSE). Moreover, we would need to perform some sort of cross-validation. While generating our regression models, for example with XGBoost, we have continuously monitored our errors on the training and test datasets to avoid overfitting. However, from an HPC perspective, the most important metric is the eventual performance improvement of the method, i.e., we would like to answer whether the machine learning effort improves the running time of our MPI collectives.

For that reason, we evaluate our approach in the following way. We select a reasonable subset of compute nodes, which will be used for the training of our regression models. The idea is that we usually get an allocation of compute nodes from the batch scheduler. When we have such an allocation, we perform MPI benchmarking runs on all compute nodes.

Table III gives an overview of the training and test datasets used in our analysis.[1] For example, on *Hydra*, we train our regression models for `MPI_Bcast` with the seven different numbers of compute nodes. Then, we apply our model on the test dataset, which in this case only includes odd numbers of compute nodes. Of course, we could have fully randomized these datasets in this paper, which we had also tested in our study. The results were very similar to the ones we present here. Still, we believe that this choice of training and test datasets is very realistic in practice. Usually, a scientist runs some MPI benchmark on a few different, but commonly used numbers

[1] https://github.com/hunsa/mpi-collective-prediction

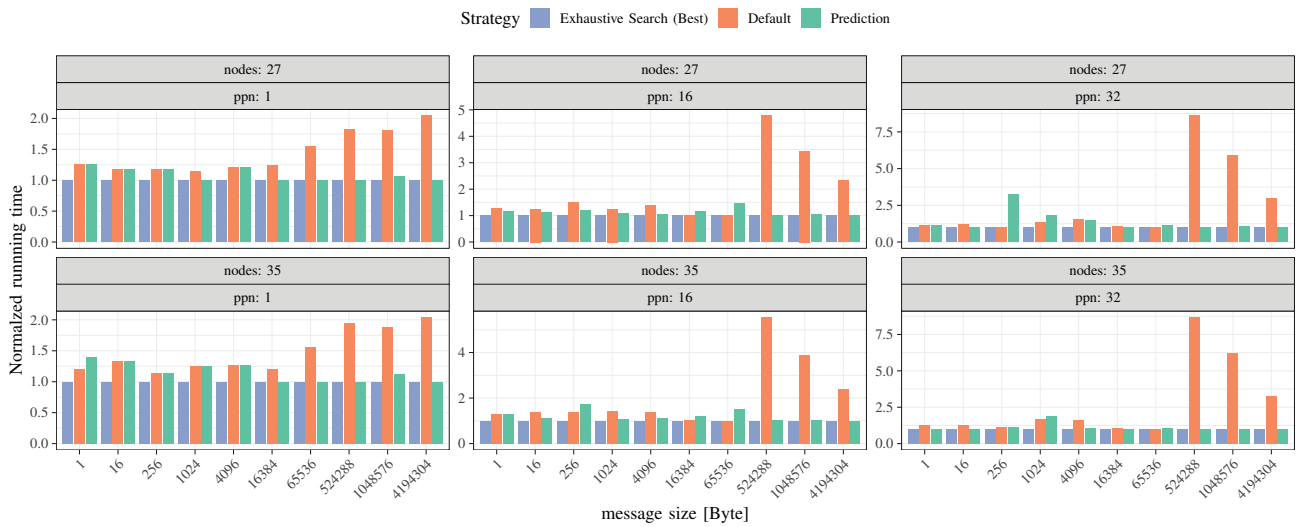

Fig. 4: Comparison of the algorithm selection strategies for `MPI_Bcast`; Open MPI 4.0.2; *Hydra*.

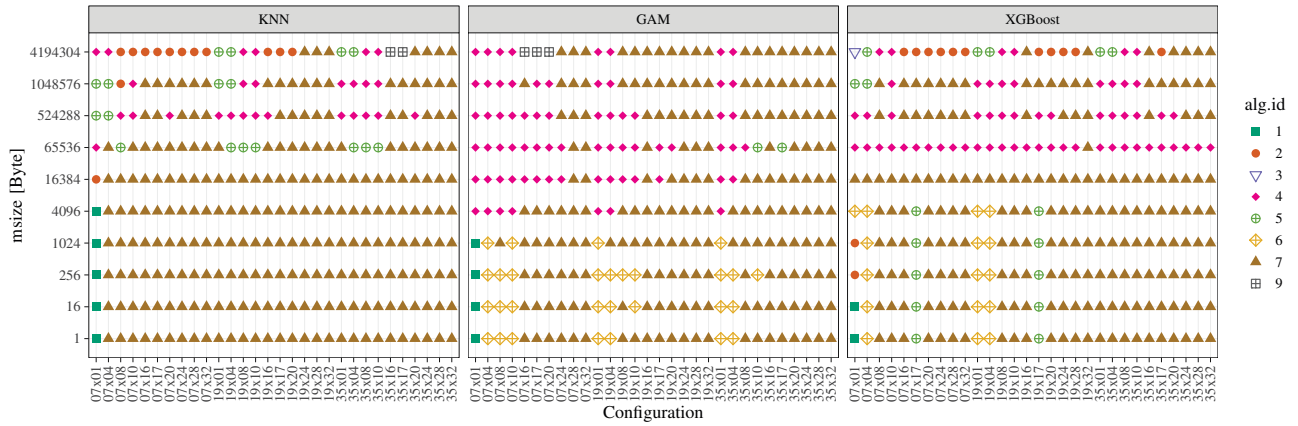

Fig. 5: Overview of the predicted algorithm for various process configurations (#nodes × ppn) for `MPI_Bcast` with each of the regression learners; Open MPI 4.0.2; *Hydra*.

of compute nodes, e.g., 8, 16, and 32, and these result should then form the foundation of the model.

We note that we have measured the performance for the entire dataset beforehand, i.e., we do not need to run benchmarks on-the-fly. If our models predict a certain algorithm and its configuration for an unseen feature vector, we already know the actual running time of that algorithm with this configuration. We also know the (empirically) best algorithm for this feature vector, which will serve as a reference point. We would like to beat the baseline, which is the default algorithm selected by the decision logic. In Open MPI, the default strategy is called algorithm 0, and in Intel MPI the decision logic is used if no explicit algorithm is set by the user.

It is also important to discuss the training time. Since we utilize ReproMPI, we configured it to benchmark each individual configuration for a maximum of either 500 values or 0.5 s on *SuperMUC-NG* and 1 s on *Hydra* and *Jupiter*, whatever condition is satisfied quicker. For small message sizes, recording 500 measurements usually takes much less than one second. For example, on *SuperMUC-NG*, the training would require a maximum of $23184 \cdot 0.5\,\mathrm{s}$, which amounts to roughly 3 hours. However, in practice, the actual measurements on *SuperMUC-NG* took only about 56 minutes on all compute nodes.

A. Prediction Results for Hydra

Finally, let us inspect the experimental results starting with *Hydra*. Due to space limitations, we can only show a subset of prediction results for Open MPI in Figure 4. Here, we compare the MPI library performance for three cases: (1) the best possible algorithm, found by an exhaustive search, (2) the default algorithm, decided by the decision logic in Open MPI, and (3) our predicted algorithm. All prediction results shown here are obtained using GAM. Since we know the running

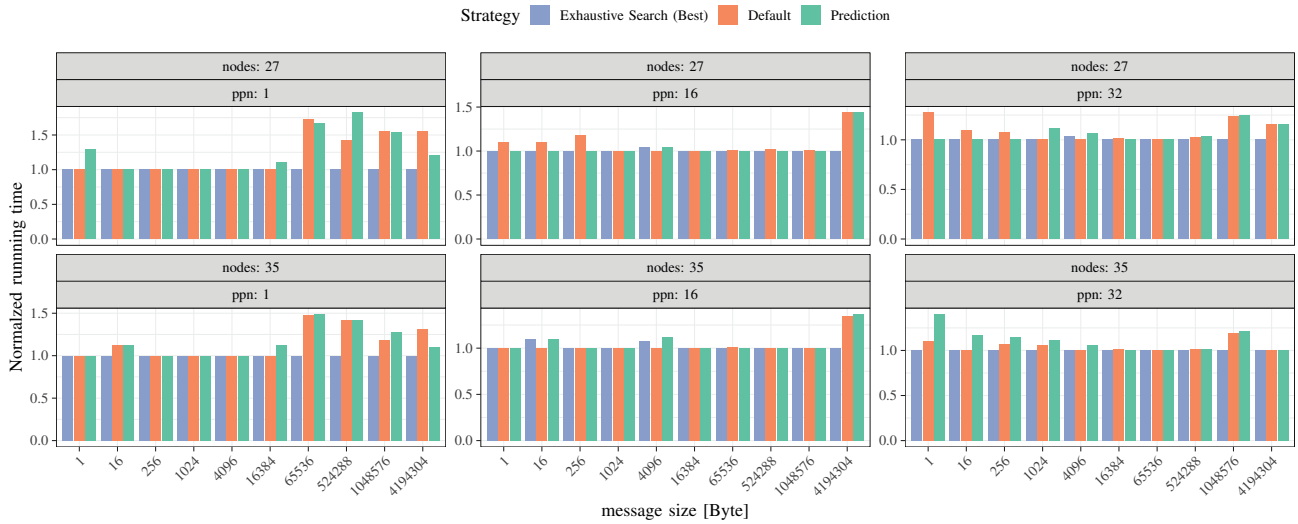

Fig. 6: Comparison of the algorithm selection strategies for `MPI_Allreduce`; Intel MPI 2019; *Hydra*.

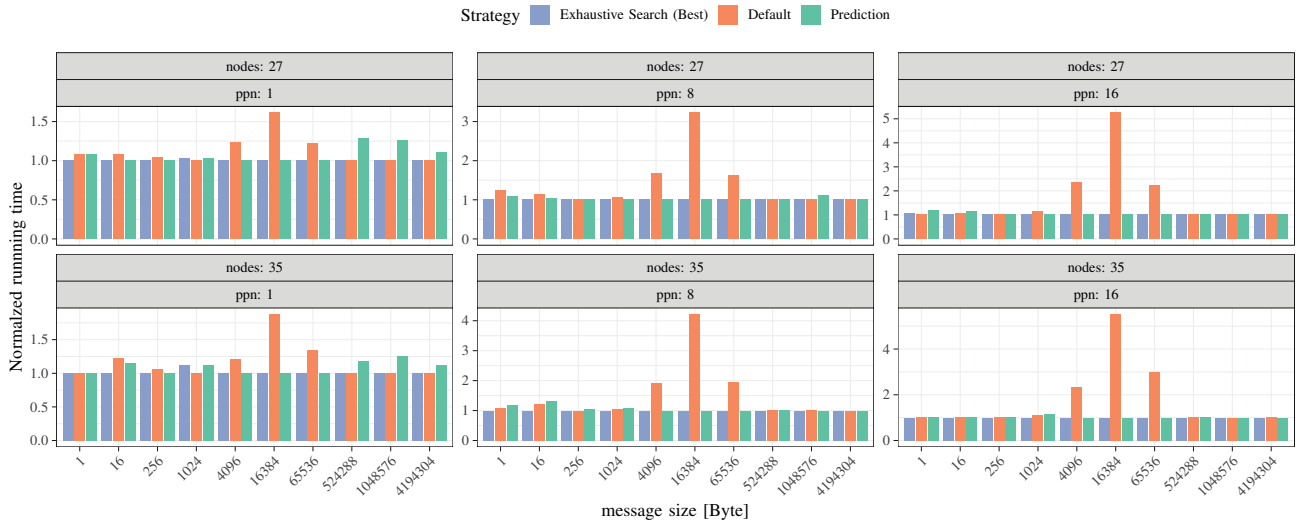

Fig. 7: Comparison of the algorithm selection strategies for `MPI_Allreduce`; Open MPI 4.0.2; *Jupiter*.

time for all three configurations, we normalize the running time with respect to the best possible. Therefore, the exhaustive search will have a normalized score of 1.0. We can observe that our predicted algorithm is very close to the best possible algorithm for most of the cases, and it clearly outperforms the default Open MPI algorithm selection strategy.

We asked ourselves whether all broadcast algorithms implemented in Open MPI were used by the predictor. Therefore, we show, in Figure 5, the selected algorithm for each process configuration (x axis) and each message size (y axis). The graph also compares the decisions taken by the different regression strategies (KNN, GAM, and XGBoost). The broadcast algorithm 8 is missing in the figure, as it was found buggy in this version of Open MPI. We can observe that all regression approaches indeed lead to a very different selection strategy and that all algorithms were used in the predictions. We note that, for the sake of clarity, we only show the algorithm IDs and omit their parameter settings such as segment size and fanout (which are part of this model but now shown).

We also analyze the prediction potential for Intel MPI in Figure 6. We notice that the default strategy of Intel MPI is very efficient, as it already selects the best algorithm in many cases. For `MPI_Allreduce`, there is no significant difference between the default and the predicted strategy. Although this looks like a weak point of our approach at first glance, it is in fact very good. Without any detailed meta-knowledge of the algorithms and their behavior, our approach is able to keep up with the decision logic provided.

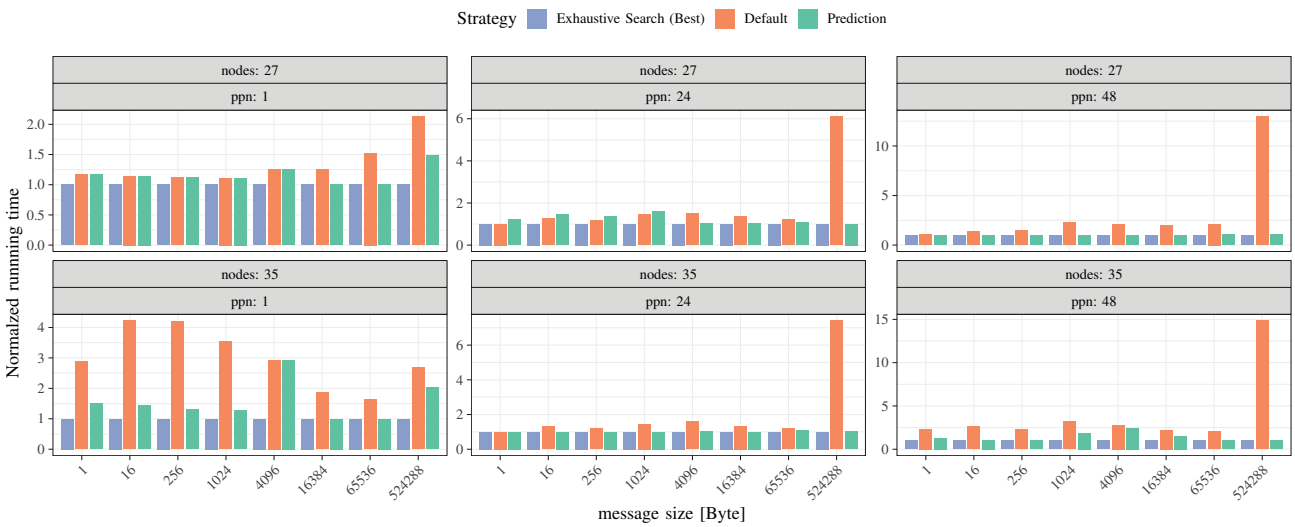

Fig. 8: Comparison of the algorithm selection strategies for `MPI_Bcast`; Open MPI 4.0.2; *SuperMUC-NG*.

TABLE IV: Overall prediction quality, measured as the relative speed-up over the default selection strategy (the higher the better).

(a) Large training dataset

method	d1	d2	d3	d4	d5	d6	d7	d8	mean
KNN	1.68	1.49	1.49	1.16	1.04	0.84	1.11	2.13	1.37
GAM	1.65	2.16	1.41	1.28	1.02	1.01	1.11	2.17	1.48
XGBoost	1.71	2.11	1.41	1.19	0.99	0.98	1.10	1.82	1.41

(b) Small training dataset

method	d1	d2	d3	d4	d5	d6	d7	d8	mean
KNN	1.68	1.45	1.43	1.11	1.03	0.84	1.11	2.13	1.35
GAM	1.67	2.16	1.41	1.21	1.02	1.00	1.11	2.17	1.47
XGBoost	1.60	1.87	1.35	1.06	0.94	0.95	0.97	1.82	1.32

B. Prediction Results for Jupiter

On *Jupiter*, the results for `MPI_Allreduce` are similar to the `MPI_Allreduce` results on *Hydra*. We specifically investigated `MPI_Allreduce` since it is the most commonly used collective according to Chunduri et al. [14]. Again, the default decision logic, this time of Open MPI, is already performing very well, as can be seen in Figure 7. Thus, there is not much to gain for most message sizes. Nonetheless, there is always a range of message sizes, around 16 kB, for which our predicted algorithm performs significantly better.

C. Prediction Results for SuperMUC-NG

The prediction results shown in Figure 8 for the *SuperMUC-NG* dataset are also promising. We can observe that our approach selects better algorithms for `MPI_Bcast` in several cases. Although we see these spikes for the largest message size, we would say that the overall performance of the default and our prediction strategy are equal.

D. Overall Evaluation

Last, we evaluate how good the prediction models are depending on the size of the training data and the learning strategy. To that end, we compute the speed-up of our predicted algorithm with respect to the default strategy. Hence, a speed-up value that is larger than one would mean that the predicted algorithm is better. We are interested in obtaining an overall, average speed-up that is as large as possible. We show the overall performance results in Table IV, where Table IVa summarizes the mean speed-up of the predicted algorithms for the large training dataset. For example, applying KNN, our prediction leads to a 37% improvement of the running time on average. We can observe that KNN, GAM, and XGBoost lead to similar results on all datasets. Interestingly, the improvements obtained for the small datasets are very similar, which means that with a moderate training effort one can already get about 30–45% improvement in running time compared to the default strategy.

VI. Related Work

Analyzing and optimizing MPI collectives has always been an active field of study [4], [5], as collective operations play an important role in many parallel, scientific applications.

An online approach to tuning MPI collectives is STAR-MPI [22], which internally selects an appropriate algorithm for a specific input instance. It works in two phases. In the tuning phase, it benchmarks different algorithms for one MPI function and records their run-times. Then, STAR-MPI has enough statistical information that it can select a good algorithm for subsequent calls of this MPI function. Chaarawi et al. [6] developed the Open Tool for Parameter Optimization (OTPO), which can be used to tune parameters of the Open MPI library. OTPO effectively performs an exhaustive search over a small number of parameters. Pellegrini et al. [23] tackled the problem of finding good MPI library parameters, such

as the eager limit, with machine learning techniques. They train different models (decision trees and artificial neural networks) based on performance features of application runs, such as the percentage of collective operations in a training run. In contrast to this work, our approach works without a previous profiling run, as all the features are already known (message size, number of processes). Barigou and Gabriel [24] showed how to tune algorithmic parameters of non-blocking collectives automatically, where parameters such as fan-out and segment size for `MPI_Ibcast` are optimized. Sikora et al. [25] presented an auto-tuning approach for MPI applications with the Periscope tool-chain. The tuning tool then either performs an exhaustive search or uses a genetic algorithm to find a good allocation of all parameters. Papadopoulou et al. [26] proposed a method based on machine-learning techniques to predict the point-to-point communication time of HPC applications, where they used additional features, such as the number of intra- or inter-node messages, to improve the prediction performance. In contrast, our approach does not rely on MPI profiling data of a run, whose execution time we try to predict, as these data are only available post mortem.

Pjesivac-Grbovic et al. [27] conducted a very detailed analysis of the applicability of different performance models for MPI collectives. In this work, the authors assessed the prediction accuracy of various performance models for different algorithms of the following collectives: Barrier, Broadcast, Reduce, and Alltoall. Each algorithm was modeled using the Hockney, the LogGP, and the P-LogP model, and the predicted performance values were compared to experimentally determined values. A similar work was lately published by Nuriyev and Lastovetsky [7].

Shudler et al. [28] have shown how to pinpoint performance problems of MPI collectives using performance models. In their approach, an empirically fitted performance model is compared to a theoretical expectation of the running time for a collective. If these models defer, a performance problem is detected. We proposed an orthogonal approach [29], where we systematically evaluated whether MPI collectives fulfill certain performance guidelines. A possible performance guideline is that an allreduce call should never be slower than chaining reduce and broadcast. PGMPITuneLib checks these performance guidelines empirically and records the cases where the default MPI algorithm violates a guideline. In a later MPI application run, PGMPITuneLib can now substitute the original MPI collective with the guideline implementation.

The Artificial Intelligence community has tackled the problems of algorithm selection [10] and algorithm configuration [11] over the last decades. A common question is how to find a good solver/algorithm for an arbitrary instance of a certain problem, e.g., Mixed Integer Programming or the Traveling Salesman Problem. Several proposed solutions iteratively learn and build an algorithmic portfolio for a set of training instances, which then works well on unseen instances [30].

Other approaches to parameter tuning include tools like OpenTuner [31], which offers a search strategy called "AUC Bandit meta technique" (an ensemble technique) that works better than other search techniques in isolation (e.g., hill-climbing or evolutionary methods).

VII. Conclusions

In this work, we revisited the algorithm selection problem for MPI collective operations. This problem consists in selecting the best possible (the fastest) algorithm and its parameters for a specific use case or scenario. A use case (or instance) is composed of the actual collective call, e.g., `MPI_Allreduce`, the message size, the number of compute nodes, and the number of processes per compute node. We proposed a novel algorithm selection strategy that builds a regression model for each algorithm and configuration. A configuration defines the setting of the various algorithmic parameters, such as the segment size or the tree fanout.

As we set out to devise a general tuning framework for MPI collectives using regression models, we evaluated our approach with different learning strategies, e.g., KNN, GAM, or XGBoost, in order to highlight the independence of our method on a specific learner or its hyper-parameters. We examined our prediction models for multiple MPI collectives on three different parallel machines and two MPI libraries (Open MPI and Intel MPI). Our experimental results support the claim that our algorithm selection approach improves the overall performance of Open MPI in all considered cases. For Intel MPI, we found that the default strategy already selects the best possible algorithm in most of the cases, which limited the tuning potential for our method. Nonetheless, our predictor performs equally well, which shows the robustness of our approach.

Acknowledgment

We acknowledge PRACE for awarding us access to SuperMUC-NG at GCS@LRZ, Germany. This work was supported by funding provided by the University of Maryland College Park Foundation.

References

[1] R. L. Graham, G. M. Shipman, B. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A high-performance, heterogeneous MPI," in *CLUSTER*. IEEE Computer Society, 2006.

[2] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The MVAPICH Project: Evolution and Sustainability of an Open Source Production Quality MPI Library for HPC," in *Proceedings of the First Workshop on on Sustainable Software for Science: Practice and Experiences (WSSSPE)*, 2013.

[3] W. Gropp, "MPICH2: A new start for MPI implementations," in *Proceedings of the 9th EuroPVM/MPI*, ser. Lecture Notes in Computer Science, vol. 2474. Springer, 2002, p. 7.

[4] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[5] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, 2005.

[6] M. Chaarawi, J. M. Squyres, E. Gabriel, and S. Feki, "A tool for optimizing runtime parameters of Open MPI," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, ser. LNCS, vol. 5205, 2008, pp. 210–217.

[7] E. Nuriyev and A. L. Lastovetsky, "Accurate runtime selection of optimal MPI collective algorithms using analytical performance modelling," *CoRR*, vol. abs/2004.11062, 2020.

[8] J. Pjesivac-Grbovic, G. Bosilca, G. E. Fagg, T. Angskun, and J. Dongarra, "MPI collective algorithm selection and quadtree encoding," *Parallel Computing*, vol. 33, no. 9, pp. 613–623, 2007.

[9] S. Hunold and A. Carpen-Amarie, "Algorithm selection of MPI collectives using machine learning techniques," in *Proceedings of the IEEE/ACM Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS@SC)*, 2018.

[10] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann, "Automated algorithm selection: Survey and perspectives," *Evolutionary Computation*, vol. 27, no. 1, pp. 3–45, 2019, pMID: 30475672.

[11] H. H. Hoos, "Automated algorithm configuration and parameter tuning," in *Autonomous Search*, Y. Hamadi, E. Monfroy, and F. Saubion, Eds. Springer, 2012, pp. 37–71.

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

[13] "Intel MPI Benchmarks." [Online]. Available: https://github.com/intel/mpi-benchmarks

[14] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI usage on a production supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE / ACM, 2018, pp. 30:1–30:15.

[15] M. Lindauer, H. H. Hoos, F. Hutter, and T. Schaub, "Autofolio: An automatically configured algorithm selector," *J. Artif. Intell. Res.*, vol. 53, pp. 745–778, 2015.

[16] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3617–3630, 2016.

[17] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*, ser. Springer Series in Statistics. Springer, 2009.

[18] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2016, pp. 785–794.

[19] T. Hastie and R. Tibshirani, *Generalized additive models*. Wiley Online Library, 1990.

[20] T. M. Mitchell, *Machine Learning*, ser. McGraw Hill series in computer science. McGraw-Hill, 1997.

[21] S. Hunold and A. Carpen-Amarie, "Hierarchical clock synchronization in MPI," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[22] A. Faraj, X. Yuan, and D. K. Lowenthal, "STAR-MPI: self tuned adaptive routines for MPI collective operations," in *Proceedings of the International Conference on Supercomputing (ICS)*. ACM, 2006, pp. 199–208.

[23] S. Pellegrini, J. Wang, T. Fahringer, and H. Moritsch, "Optimizing MPI runtime parameter settings by using machine learning," in *EuroPVM/MPI*, ser. LNCS, vol. 5759. Springer, 2009, pp. 196–206.

[24] Y. Barigou and E. Gabriel, "Maximizing communication-computation overlap through automatic parallelization and run-time tuning of non-blocking collective operations," *Int. J. Parallel Program.*, vol. 45, no. 6, pp. 1390–1416, 2017.

[25] A. Sikora, E. César, I. A. C. Ureña, and M. Gerndt, "Autotuning of MPI applications using PTF," in *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*. ACM, 2016, pp. 31–38.

[26] N. Papadopoulou, G. I. Goumas, and N. Koziris, "Predictive communication modeling for HPC applications," *Cluster Computing*, vol. 20, no. 3, pp. 2725–2747, 2017.

[27] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[28] S. Shudler, Y. Berens, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf, "Engineering algorithms for scalability through continuous validation of performance expectations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1768–1785, 2019.

[29] S. Hunold and A. Carpen-Amarie, "Autotuning MPI collectives using performance guidelines," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia)*. ACM, 2018, pp. 64–74.

[30] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artif. Intell.*, vol. 206, pp. 79–111, 2014.

[31] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *PACT*. ACM, 2014, pp. 303–316.