Algorithm Selection of MPI Collectives using Machine Learning Techniques

Sascha Hunold
TU Wien, Faculty of Informatics
Vienna, Austria
Email: hunold@par.tuwien.ac.at

Alexandra Carpen-Amarie
Fraunhofer ITWM
Kaiserslautern, Germany
Email: alexandra.carpen-amarie@itwm.fraunhofer.de

***Abstract*—Autotuning is a well established method to improve software performance for a given system, and it is especially important in High Performance Computing. The goal of autotuning is to find the best possible algorithm and its best parameter settings for a given instance. Autotuning can also be applied to MPI libraries, such as Open MPI or Intel MPI. These MPI libraries provide numerous parameters that allow users to adapt them to a given system. Some of these tunable parameters enable users to select a specific algorithm that should be used internally by an MPI collective operation. For the purpose of automatically tuning MPI collectives on a given system, the Intel MPI library is shipped with `mpitune`. The drawback of tools like `mpitune` is that results can only be applied to cases (e.g., number of processes, message size) for which the tool has performed the optimization.**

To overcome this limitation, we present a first step towards tuning MPI libraries also for unseen instances by applying machine learning techniques. Our goal is to create a classifier that takes the collective operation, the message size and communicator characteristics (number of compute nodes, number of processes per node) as an input and gives the predicted best algorithm for this problem as an output. We show how such a model can be constructed and what pitfalls should be avoided. We demonstrate by thorough experimentation that our proposed prediction model is able to outperform the default configuration of Intel MPI or Open MPI on recent computer clusters.

***Index Terms*—MPI, Intel `mpitune`, performance prediction, autotuning, MPI benchmarks, collective communication operations, machine learning**

I. Introduction

In this paper, we consider the algorithm selection problem in the context of the Message Passing Interface (MPI). The goal is to select an efficient algorithm for a particular MPI collective operation when given a set of inputs, which consist of the message size, the number of compute nodes, and the number of processes per compute node. The problem arises when one attempts to tune a particular MPI library, e.g., Open MPI or Intel MPI, to a given parallel machine. MPI libraries usually provide different implementations of MPI collectives for various use cases and contain some logic to select a decently performing implementation. However, as the search space is tremendous (Open MPI has hundreds of possibly dependent parameters), a library should be adapted or automatically tuned to a specific machine. Open MPI and Intel MPI provide tools that help completing this task. For example, Intel MPI is shipped with the `mpitune` tool, and the Open MPI website names `otpo` [1] as the tool to use for tuning Open MPI parameters. Both tools, however, essentially conduct an exhaustive search of the restrained search space defined by the user.

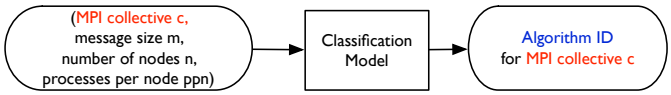


Fig. 1: Classification Problem for MPI Collectives.

Now, the question is whether we can use this information about the performance of MPI collectives gathered for specific communicators and message sizes to predict a good algorithm for unseen instances. The problem can be treated as a classification problem, which is depicted in Figure 1. We are given a collective operation, a communicator, and a message size, and want to predict the best possible algorithm for this communication problem. To accomplish that, we first run a series of measurements for specific and relatively common configurations (e.g., deploy one MPI process per compute node or deploy as many MPI processes as cores on a compute node). Now that the best algorithm for each configuration is known, we can learn a prediction model and apply it to select algorithms for unseen instances. In this work, we show how such a prediction model can be constructed and point out several pitfalls when building such a classification model. To demonstrate that our approach works in practice, we also conduct a series of experiments that compare the tuned and default version of MPI collectives. Although we apply the selection strategy offline, the prediction model could also be applied from within an MPI library. However, since the Intel MPI library does not provide an API to change algorithms during the run-time of MPI applications, we currently only generate configuration files that can be loaded when starting an MPI application.

The paper is structured as follows. We first motivate our work by showing an example scenario for Intel MPI in Section II. We also discuss related work that considers learning techniques for selecting appropriate algorithms or parameters for MPI libraries and applications. In Section III, we present our approach to learn a prediction model for MPI collectives. We show experimental results in Section IV and conclude with Section V.

II. Related Work and Background

A. Motivational Example

The Intel MPI library is bundled with the `mpitune` tool, which can be used to tune the library on a given system. Besides general one-sided or two-sided communication parameters, such as the eager threshold, `mpitune` can also tune (select appropriate) algorithms for MPI collective operations. For that purpose, it runs the Intel MPI Benchmarks for a predefined number of communicators, always given as a tuple (number of computer nodes, number of processes per node). `mpitune` (with the help of the Intel MPI Benchmarks) measures the run-time of every algorithm that is provided for a collective operation. With this performance information, `mpitune` generates a configuration file with options for each collective. An example configuration file is shown in Listing 1.

Listing 1: Example content of an Intel MPI configuration file.

```
-genv I_MPI_ADJUST_BCAST '11:1-4;10:5-16;11:17-61;
    10:62-107;1:108-701;10:702-16384;11:16385-32768;
    9:32769-524288;11:524289-2147483647'
-genv I_MPI_ADJUST_ALLGATHER '1:1-2;5:3-4;2:5-191;
    1:192-479;5:480-512;4:513-3828;
    3:3829-2147483647'
```

The individual lines of the configuration file define for which message size a certain algorithm should be used. It is noticeable that each line covers the entire message range from $1\,\mathrm{Byte}$ to `MAX_INT` bytes. Since the Intel MPI Benchmarks measure the run-time of collectives only for power-of-two message sizes, `mpitune` applies the following interpolation heuristic to cover the entire message range: if algorithm A is the best for x bytes and algorithm B for $x+z$ bytes, it will use algorithm A for the interval $[x, x+z/2]$ and algorithm B for the interval $(x+z/2, x+z]$.

We believe that there are two possible ways to improve this method of tuning collectives:

1) Since a tuned configuration file can currently only be applied for the matching input tuple of (number of nodes, processes per node), we want to exploit the already recorded run-time information to predict the best performing algorithm for unseen cases.
2) Only measuring for powers of two for either message size or communicator size can often be misleading [2, Sec.4.2], and thus, testing with other message sizes can possibly lead to a more robust tuned configuration. The simple interpolation method of `mpitune` could be improved, especially as it relies on powers-of-two measurements.

In the present article, we will focus on the first aspect to improve MPI libraries.

B. Tuning and Predicting MPI Libraries

The work of Pješivac-Grbović et al. [3] is closely related to our proposed method. They examined whether decision trees can capture the performance characteristics of different MPI collectives for a range of message and communicator sizes. The authors worked on a classification problem to predict the best algorithm for a given pair of communicator and message size. An exhaustive search was conducted to collect the training data set, which was then used to construct decision trees with different depths. These decision trees were finally compared by their classification errors on the given data set. While they focus on answering whether quadtrees can be used for generating decision functions, our work applies tree-based classification techniques to generate useful algorithm predictions for MPI collectives, and we provide a framework to build tuning models for a specific machine and apply them for unseen configurations.

Pellegrini et al. [4] also relied on machine learning techniques to tune the parameters of MPI libraries. Instead of tuning the algorithm selection, they focused on tuning general MPI parameters, such as the Open MPI parameters *mpi_preconnect_mpi* or *eager_limit*. In this work, they used decision trees and neural networks to predict good parameter settings for several NAS Parallel Benchmarks. Later, they also showed how ANOVA can help to find parameters with the largest effect on the application run-time [5].

In the context of autotuning MPI collectives, we presented the PGMPITuneLib [6], which tunes MPI libraries by finding a better performing replacement of an MPI collective operation. In this approach, it is first checked if an MPI library fulfills obvious performance guidelines, such as `MPI_Allreduce` $\preceq$ `MPI_Reduce` + `MPI_Bcast`, which means that `MPI_Allreduce` should not be slower than its straightforward replacement. If it is slower, PGMPITuneLib fixes this performance deficit by replacing the original call with the best performing semantically equivalent variant. Additionally, we showed that these new variants can be faster than already implemented algorithms, all of which had been tuned with OpenTuner [7] to find their best possible parameter configuration. However, PGMPITuneLib is not a prediction framework, but it could be combined with the approach shown in this paper to predict a better performing replacement of the algorithms selected by an MPI library for a specific case.

III. MPI-CP: Prediction Model for MPI Collectives

It is our goal to obtain a prediction model for the following classification problem:

$$f(c, m, n, ppn) \rightarrow alg\_id, \tag{1}$$

where c, m, n, and ppn denote the MPI collective operation, the message size, the number of compute nodes, and the number of processes per node, respectively. Since we are primarily interested in a proof-of-concept, the present paper does not attempt to find the best classifier for the MPI algorithm selection problem. That said, we also do not consider any hyperparameter tuning of the actual learning process, and thus, we are mainly looking for a reasonably simple and understandable model. For that reason, we started with decision tree learning. We note that we have also tried learning with neural networks, but with such a small feature vector, choosing the right input scaling technique becomes complex and, in general, we needed hyperparameter tuning to either converge or to get useful predictions.

TABLE I: Search space for model building.

MPI collectives	`MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Reduce`, `MPI_Alltoall`, `MPI_Allreduce`, `MPI_Allgather`
message sizes m	$2^i : 0 \leq i \leq 20$
number of nodes n	$2, 8, 12, 16, 24, 36$
processes per node ppn	$1, 16, 32$

A. Step 1: Building a Classification Model

In our first attempt, we ran a series of benchmarks and varied the parameters shown in Table I. We identified the best algorithm for each individual case. The best algorithm for each case is then used with its corresponding feature vector to train the model. Next, we attempted to directly build a classifier to predict the best algorithm for any unseen case (predicting a class). The training data set looks like this:

	test	n	ppn	msize	algid
1	MPI_Bcast	16	1	1	8
2	MPI_Bcast	16	1	2	11
3	MPI_Bcast	16	1	4	8
4	MPI_Bcast	16	1	8	8
5	MPI_Bcast	16	1	16	8
6	MPI_Bcast	16	1	64	8
7	MPI_Bcast	16	1	128	10

The problem with this approach is that we will experience a class bias, as not all classes have the same (or a similar) number of training samples. In most cases (unlike the example shown above), the default algorithm (0) was the fastest for the majority of samples in the training data set, leaving only a small subset of cases in which one of the other algorithms had a better performance. As off-the-shelf decision tree methods are known to have problems with such imbalanced input data sets, we had to change our approach.

B. Step 2: Building a Regression Model

The idea is to build a regression model of the relative performance of each algorithm compared to the default algorithm (0). Instead of building only one model that can directly predict the algorithm ID for an unseen case, we generate k different prediction models, where k is the number of available implementations for a given MPI collective (one model for each algorithm ID). Our input data sets look now similar to the following table:

	test	n	ppn	msize	algid	tmedian	normtime
1	MPI_Bcast	16	1	1	0	5.25	1.00
2	MPI_Bcast	16	1	1	1	5.72	1.09
3	MPI_Bcast	16	1	1	10	4.53	0.86
4	MPI_Bcast	16	1	1	11	5.48	1.05
5	MPI_Bcast	16	1	1	2	8.82	1.68
6	MPI_Bcast	16	1	1	3	24.32	4.64
7	MPI_Bcast	16	1	1	4	5.96	1.14

The column `tmedian` denotes the median run-time for one set of parameters (one feature vector), while `normtime` denotes the relative run-time of an algorithm with respect to algorithm 0. With such an approach each model is built with the same number of training samples for all algorithms. Nevertheless, we still have a classification problem to solve and want to predict exactly one final class (the algorithm ID). To do that, we predict each algorithm's relative performance using the k regression models. The algorithm with the lowest relative performance (equivalent to the highest speedup) is selected. This algorithm selection process is illustrated in Figure 2.

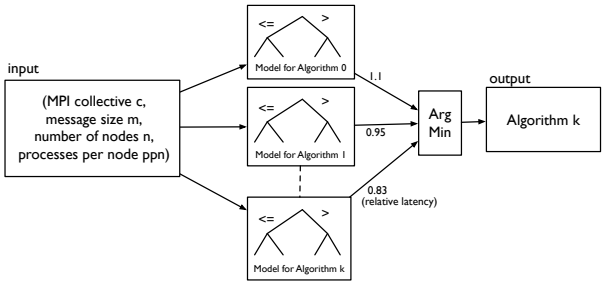


Fig. 2: Strategy for the algorithm selection of MPI collectives.

C. Step 3: Data Transformation

Predicting the algorithm ID based on the relative run-time worked reasonably well. But we noticed a serious flaw. For some collectives, e.g., `MPI_Allreduce`, most of the implementations performed significantly worse than the default algorithm. But for some particular message sizes, one of the algorithms was always faster than the default algorithm. The problem was that, in the final decision trees, the information that this algorithm would be good in these cases was not preserved. The reason is that our decision variable, the relative run-time, is not scale-free, and splitting methods like decision trees will be affected by a distribution skewed towards larger input values. Our current solution is to transform the data by trimming down the relative run-time. As we are only interested in learning the cases where an algorithm is a good replacement candidate, we change the `normtime` to 1.01 for all algorithms that are slower than the default implementation. The resulting input data set looks eventually like this:

	test	n	ppn	msize	algid	tmedian	normtime
1	MPI_Bcast	16	1	1	0	5.25	1.00
2	MPI_Bcast	16	1	1	1	5.72	1.01
3	MPI_Bcast	16	1	1	10	4.53	0.86
4	MPI_Bcast	16	1	1	11	5.48	1.01
5	MPI_Bcast	16	1	1	2	8.82	1.01
6	MPI_Bcast	16	1	1	3	24.32	1.01
7	MPI_Bcast	16	1	1	4	5.96	1.01

With this approach, the resulting decision trees can predict cases for which faster algorithms exist (prediction of speedup) better than before, but the models cannot be used to predict accurate slowdowns.

IV. Experimental Evaluation

A. Experimental Setup

Originally, we set out to use `mpitune` throughout this process, but it turned out to be far too slow. As mentioned previously, `mpitune` uses the Intel MPI Benchmarks to conduct the measurements. The problem with benchmarking tools like the Intel MPI Benchmarks or OSU Micro-Benchmarks is that the user needs to specify a number of iterations that will be executed for each message size. The default values used in the Intel MPI Benchmarks are simply too big for larger message sizes, and thus, `mpitune` takes very long to complete (after 10

TABLE II: Hardware used in our experiments.

Name	Hardware	MPI Libraries
Hydra	36 × Dual Intel Xeon Gold 6130 @ 2.1 GHz, Intel OmniPath	Intel MPI Library 2018.3.222 Open MPI 3.1.0

hours only a fifth of the tuning process was completed on our 36-node cluster). Theoretically, one can use another benchmark tool inside `mpitune`, but this tool would need to produce the exact same output format as the Intel MPI Benchmarks.

Therefore, we implemented our own tuning framework on top of our ReproMPI benchmark [8], which features the Round-Time [9] measurement scheme. The Round-Time scheme allows measuring for a predefined period of time instead of using a fixed number of iterations, which makes the execution time of the tuning process predictable. Moreover, Round-Time uses a logical, global clock to improve the synchronization of processes before starting a measurement, which is especially important when measuring with small message sizes [9]. In the experiments, we recorded the run-time of every MPI collective with all message sizes shown in Table I using Round-Time with a 3 s time frame. That means that we performed as many measurements for a particular message size as possible until we ran out of our allocated time frame of 3 s. All experiments were conducted on *Hydra* (cf. Table II).

B. Software

The prediction models were built using the `R` packages `rpart` (4.1-13) and `randomForest` (4.6-14). We initially tested simple decision trees (with `rpart`), but eventually applied random forests in the final experiments. The random forest models were slightly more accurate than the decision tree models, as the latter often pruned too many sub-trees. As said before, we were mainly interested in answering whether a simple learning technique can help tuning MPI libraries, which can be affirmed for both learning methods.

C. Results

Figures 3 and 4 compare the run-times measured with ReproMPI for the default and for the tuned version of Intel MPI. The graphs on the left-hand side show the absolute performance in microseconds (in a log-log plot), whereas the ones on the right-hand side present the relative performance of the tuned library compared to the default version. All experiments have been repeated five times (5 `mpiruns`), and the median of the five resulting median run-times is reported. Error bars in the plots represent the minimum and the maximum over these five median run-times. We show performance results for four different configurations ($n \times ppn$): 9×20, 17×27, 23×27, 35×17 processes. As can be seen in Figure 3a, the tuned version of Intel MPI outperforms the default version for `MPI_Bcast`. Similar performance results have been recorded for other combinations of the number of compute nodes and processes per node. Especially for large message sizes, the predicted broadcast algorithms outperformed the default variants. Yet, there is one case, `MPI_Bcast` with 8 B on 9×20 processes, where the predicted algorithm leads to a slowdown of almost 20 %. For such small payloads, the run-time is very short. When we look at the absolute differences, we see that the default algorithm is about 2 µs faster than the "tuned" version, which seems negligible for large-scale applications. To put it into perspective: our model is able to predict an algorithm that is equally fast or faster in $4 \times 20 - 1 = 79$ out of 80 cases, whereas the model leads to one misprediction. We therefore believe that this is a very promising result for decision-tree-based learning processes.

Figure 4 shows similar results for `MPI_Allgather`. In contrast to `MPI_Bcast`, the model predicted a different algorithm (other than 0) for only a small range of message sizes, which is 256 B to 1024 B. We noticed that such patterns—where one or two different algorithms are predicted for a few message sizes—occur rather frequently compared to cases like `MPI_Bcast`, where a different algorithm can be found for almost all message sizes. We can observe one case (2048 B on 23×27 processes), where our prediction is not advantageous, but yet again, how to avoid these cases by improving the learning process could be the subject of future research.

Figure 5 shows that our method of predicting a good algorithm also works for other MPI libraries like Open MPI. As we can see from these graphs, our model is able to overcome the performance deficits of the default algorithm for message sizes ranging from 128 B to 512 B.

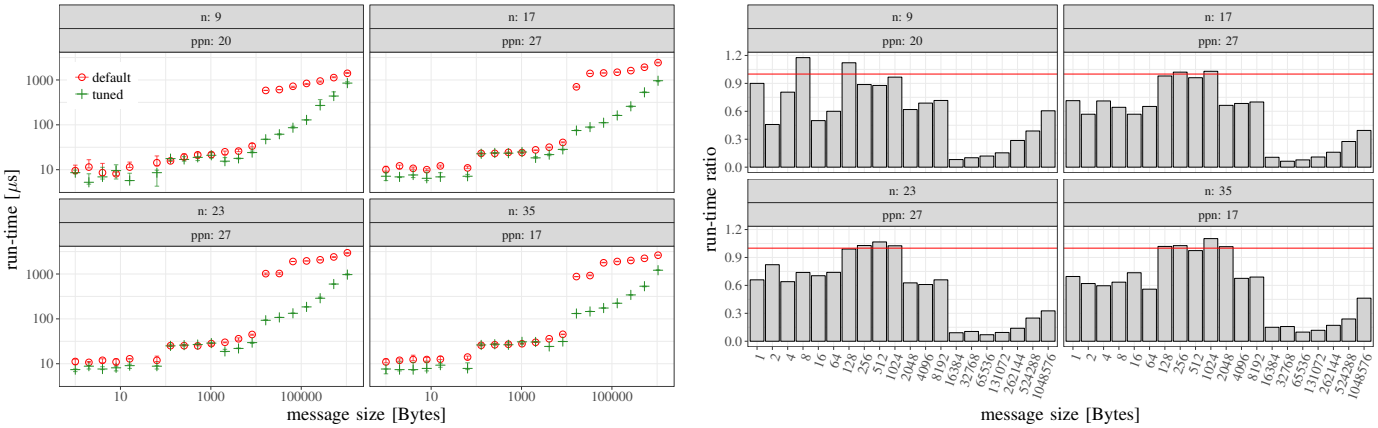
V. Conclusions and Future Work

We considered the algorithm selection problem for MPI collectives. Our goal was to learn a classifier in order to predict the best possible algorithm for a given communication problem, consisting of an MPI collective, a message size, a number of compute nodes, and the number of processes per node. We showed how to build these models using decision trees or random forests and pointed out necessary data transformations to avoid prediction pitfalls.

The experiments showed that our classifier is able to predict a better algorithm for many unseen instances, most prominently for `MPI_Bcast`. Concerning the other MPI collectives, we can state that although the default algorithm was found to be the fastest in most cases in the training data, our model was still able to predict a better algorithm in some cases (but less frequently than for `MPI_Bcast`). The important finding is that relatively simple machine learning techniques like decision trees or random forests are powerful enough to predict algorithms that can outperform the default variants.

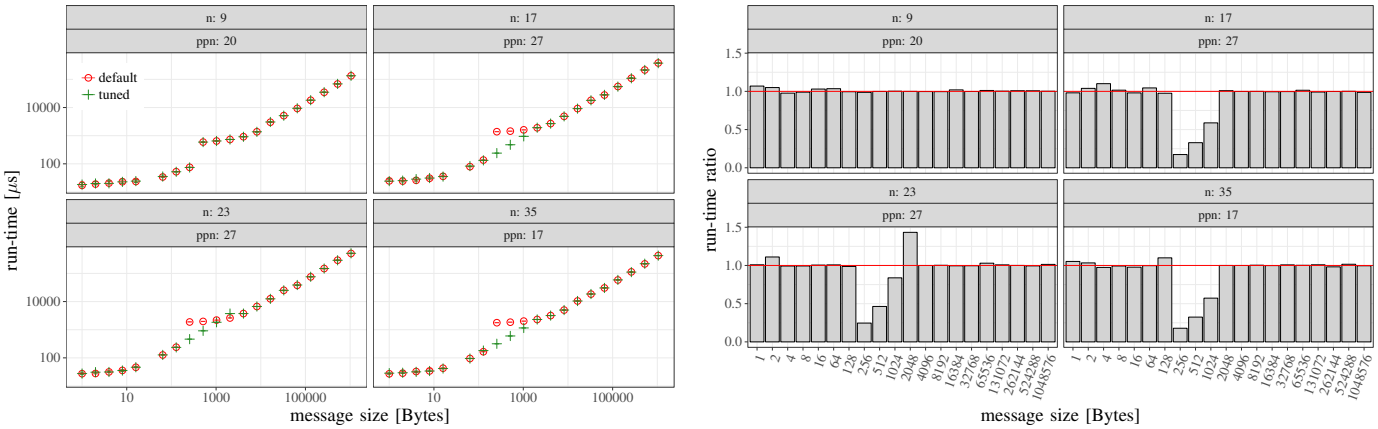
Our developed tool can load benchmark results from Intel MPI Benchmarks, OSU Micro-Benchmarks, and ReproMPI, and can produce configuration files containing the predicted algorithms (and their ranges) for Intel MPI and Open MPI.

Currently, our feature vectors and model do not contain algorithm-specific features, such as the segment size or the fan out. Since these algorithmic parameters can have a significant impact on the performance of collectives, we will consider more features in future work. We will also validate our approach on other, larger machines.



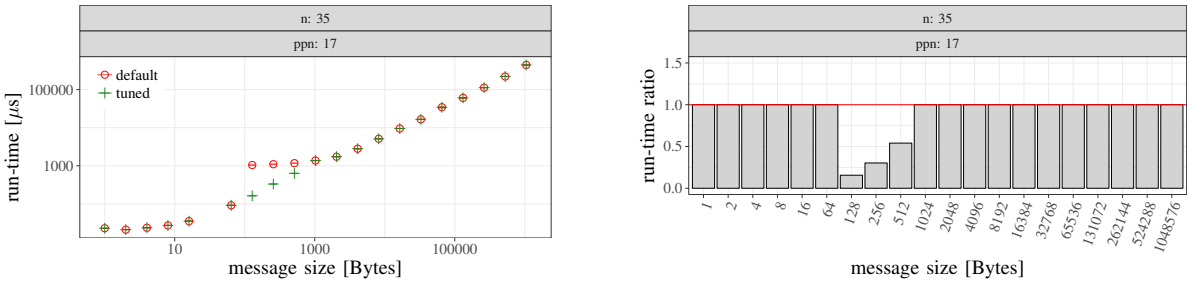
(a) absolute (b) relative

Fig. 3: Run-time of `MPI_Bcast` for various message sizes; Intel MPI Library 2018.3.222, *Hydra*, 5 `mpiruns`.



(a) absolute (b) relative

Fig. 4: Run-time of `MPI_Allgather` for various message sizes; Intel MPI Library 2018.3.222, *Hydra*, 5 `mpiruns`.



(a) absolute (b) relative

Fig. 5: Run-time of `MPI_Allgather` for different message sizes; Open MPI 3.1.0, *Hydra*, 5 `mpiruns`.

REFERENCES

[1] 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.

[2] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015, pp. 73:1–73:12.

[3] 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.

[4] 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.

[5] S. Pellegrini, R. Prodan, and T. Fahringer, "Tuning MPI runtime parameter setting for high performance computing," in *Proceedings of IEEE International Conference on Cluster Computing, Workshops*, 2012, pp. 213–221.

[6] 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.

[7] 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 *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2014, pp. 303–316.

[8] 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.

[9] ——, "Hierarchical clock synchronization in MPI," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.