OMPICollTune: Autotuning MPI Collectives by Incremental Online Learning

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

Sebastian Steiner
TU Wien, Faculty of Informatics
Vienna, Austria
steiner@par.tuwien.ac.at

***Abstract*—Collective communication operations, such as Broadcast or Reduce, are fundamental cornerstones in many high-performance applications. Most collective operations can be implemented using different algorithms, each of which has advantages and disadvantages. For that reason, MPI libraries typically implement a selection logic that attempts to make good algorithmic choices for specific problem instances. It has been shown in the literature that the hard-coded algorithm selection logic found in MPI libraries can be improved by tuning the collectives in a separate, offline micro-benchmarking run.**

In the present paper, we go a fundamentally different way of improving the algorithm selection for MPI collectives. We integrate the probing of different algorithms directly into the MPI library. Whenever an MPI application is started with a given process configuration, i.e., the number of nodes and the processes per node, the tuner, instead of the default selection logic, finds the next algorithm to complete an issued MPI collective call. The tuner records the runtime of this MPI call for a subset of processes. With the recorded performance data, the tuner is able to build a performance model that allows selecting an efficient algorithm for a given collective problem. Subsequently recorded performance results are then used to update the performance model, where the probabilities for selecting an algorithm are adapted by the tuner, such that slow algorithms get a smaller chance of being selected. We show in a case study, using the ECP proxy application miniAMR, that our approach can effectively tune the performance of Allreduce.

***Index Terms*—HPC, MPI, autotuning, algorithm selection, collective communication, machine learning**

I. Introduction

Virtually all large-scale, parallel applications running on supercomputers use the Message Passing Interface (MPI) for data communication over the network [1]. Collective communication operations, where a group of processes collectively solves a communication task, are responsible for a significant fraction of the overall application running time. Since the MPI standard only defines the semantics of collective operations, MPI libraries may use different algorithms for implementing collective operations. For example, typical algorithmic choices for realizing a Broadcast are the binomial or the binary tree algorithm. Since a binary tree uses a fixed communication pattern in each round (each process always has the same children), it is a good candidate for pipelined Broadcast algorithms. Hence, we also need to find a good segment size to optimize the pipelined binary-tree performance. For that reason, when choosing an efficient algorithm for a collective operation, we have to solve an algorithm selection and an algorithm configuration problem [2].

Prior work on collective tuning has mainly focused on static, offline tuning of MPI collectives (cf. §II). Selecting a good algorithmic candidate was based on performance results that were obtained through micro-benchmarking of MPI collectives.

In the present paper, we use a fundamentally different approach to tuning MPI collectives, as we tune an algorithm selection model iteratively. Instead of collecting the performance data by executing MPI micro-benchmarks, we use MPI application runs to retrieve performance statistics of MPI collectives. The collected performance data is used to train a performance model that reflects the quality of algorithms for specific configurations. Then, our performance model can be used either in a feedback loop for collecting more training data by running and training with more MPI applications on the target machine. It is also possible to query the performance model at any point in time to select the best possible algorithm for each configuration known so far. A configuration consists of a range of message sizes, a range of compute nodes, and a range of the processes per node, e.g., ($10\,\mathrm{B}$ to $28\,\mathrm{B}$, 4 to 8 compute nodes, 1 to 16 processes per node).

We present our online tuning approach as an extension to the tuned collective module of Open MPI 4.1.x. We have modified Open MPI such that we can intercept the default selection strategy for collective calls and then randomly select a specific, possibly different algorithm (following a given probability distribution), whose runtime we can measure inside of the application. Since we limit the number of measurements taken per application run and the number of processes that record the actual measurements, the performance datasets are relatively small and recording performance data entails a very low overhead, compared to a run without the online tuning infrastructure. Our goal is to replace the selection logic of MPI libraries with our trained and tuned model, especially for the frequently used collectives Allreduce, Broadcast, Allgather(v), and Alltoall(v).

This paper makes the following contributions:

- We present the first-of-its-kind iterative, online tuning method for MPI collective communication that reuses

This work was partially supported by the Austrian Science Fund (FWF): project P 33884.

performance data to update (train) the selection logic for choosing collective algorithms on a case-by-case basis.
- We demonstrate that our method can effectively improve the selection logic for `MPI_Allreduce` by iteratively updating our collective performance model using the ECP proxy application miniAMR.

II. Related Work

The problem of selecting a well-performing collective algorithm has already been tackled before. A pioneering work was presented by Pjesivac-Grbovic et al. [3], where decision-tree models were trained using offline micro-benchmarking. These decision trees were then translated into actual code, which are still prevalent in Open MPI.

The Intel MPI library comes with a set of tuning tools [4]. One tool can be used to exhaustively search the best algorithm for a given set of nodes and message sizes. The performance data for this optimization is gathered via micro-benchmarking. The Intel library also provides an autotuning mode with an application focus. One can start an MPI application with this autotuner, which will test different algorithms for the occurring collectives. After such a tuning run, the library selects the best performing algorithms and stores them in a tuning file, which can be used in subsequent runs. Unfortunately, there is no further information on how the process works internally.

Faraj et al. [5] presented an online tuning method called STAR-MPI. This library implements a set of algorithms for different collectives and tries to find the best-performing algorithm during the runtime of an application. Similarly to Intel MPI and our approach, Faraj et al. intercept a collective call and replace it with their own algorithm to execute the pending collective operation.

Another strategy for selecting collective algorithms is to rely on fitted analytic models, i.e., parameters like the network latency and the bandwidth are part of the model and determined experimentally on the target hardware. In a recent work, Nuriyev and Lastovetsky [6] present an analytical performance model to select well-performing algorithms for executing `MPI_Bcast` calls.

Hunold and Carpen-Amarie [7] showed how the validation of MPI performance guidelines can be used to tune MPI libraries. Performance guidelines state that a specialized MPI collective should never be slower than its re-implementation using the composition of other collectives. For example, an Allreduce call should never be slower than chaining Reduce and Bcast. The authors exploit the knowledge of guideline violations to tune an MPI library. In cases where the re-implementation was found to be faster, the selection logic uses these chained calls to improve the performance.

Last, we would like to mention two recent works that use machine learning techniques to tune the selection logic within an MPI library. Hunold et al. [8] have used micro-benchmarking results obtained for the most commonly used number of compute nodes and message sizes to build a prediction model, which allows to predict the performance of the various algorithms for unseen configurations. A disadvantage of this approach is that, depending on the predefined set of message sizes, numbers of compute nodes, types of collectives, etc., the set of required benchmark runs is relatively large, and collecting the performance dataset may take long.

To overcome this problem, Wilkins et al. [9] have proposed the ACCLAiM system, which tunes the collective calls at the beginning of a compute job. Once a batch job has been allocated, ACCLAiM uses the Jacknife method to select training points to tune collectives before the actual application starts. Tuning in the same batch job has the advantage of tuning exactly with the same compute node allocation that the application will run on.

III. Online and Iterative Tuning Approach

Now, we sketch our novel tuning approach, where a *tuning configuration* (TC) is a central cornerstone.

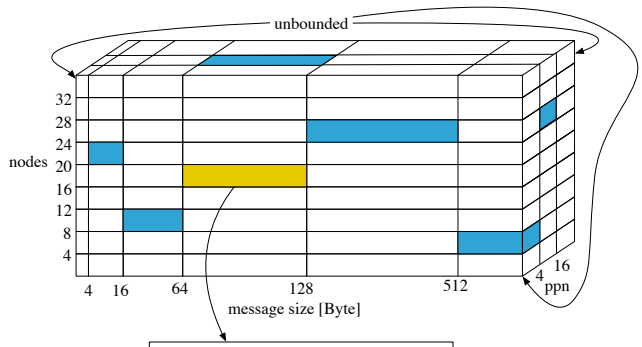
Definition 1. *Tuning Configuration. A tuning configuration is defined by a 3D block of ranges, consisting of (1) a message size range, (2) a range for the number of compute nodes, and (3) a range for the number of processes per node.*

a) The Tuning Space: In the beginning of the tuning process, the number and the size of the tuning blocks have to be defined by an HPC expert. The overall tuning space is shaped like a cuboid, which is split into individual tuning configurations (smaller cuboids), see Figure 1. This figure only represents a small example of TCs, e.g., we stop grouping message sizes after $512\,\mathrm{B}$, whereas in a real-world tuning run, we would use more ranges of message sizes. Notice that the ends of the large cuboid represent infinity, i.e., the TC cells rights of $512\,\mathrm{B}$ hold all performance data for message sizes that are strictly larger than $512\,\mathrm{B}$. We use such a cuboid of TCs for each tunable collective. Within one TC cell (bottom of Figure 1), we collect performance data and compute the probabilities for each individual algorithm.

b) Combining Algorithm Selection and Algorithm Configuration: As mentioned before, a collective algorithm, such as the binary-tree Broadcast, may have additional parameters that can impact its performance in a TC. Therefore, we apply the same strategy as Hunold et al. [8] and combine the algorithm selection and algorithm configuration by creating a larger set of possible algorithms. This is shown by example in Figure 1. For a particular collective, say Allreduce, we internally use a unique set of algorithm IDs (AlgIDs) to identify a specific algorithm and its configuration in the MPI library. For example, AlgIDs 5 and 6 in the figure are mapped to the same internal algorithm of the MPI library, which happens to be library algorithm 5. However, our AlgID 5 denotes the library algorithm 5 with segment size $1\,\mathrm{kB}$, whereas AlgID 6 also refers to library algorithm 5 but uses a segment size of $1\,\mathrm{MB}$.

We now introduce the online tuning strategy by looking at the individual steps depicted in Figure 2.

c) What to Measure?: When tuning MPI libraries offline in a micro-benchmarking run, we have to answer the important question of which message sizes or which numbers of compute nodes are most relevant for applications on the target system.



AlgID	OMPI ID	SegSize	Prob
1	1	0	0.1
2	2	0	0.3
3	3	0	0.05
4	4	0	0.15
5	5	1KB	0.1
6	5	1MB	0.1
7	5	8MB	0.1
8	6	0	0.1

Fig. 1: Performance model for tuning MPI collectives. Each cell represents a specific configuration, and the model predicts the fastest algorithm for every cell.

Our tuning approach allows for an easier solution to this problem, as we measure the runtimes of collectives exactly for the application configuration that we indeed use on the target machine. We modify the MPI library such that a collective call can be intercepted by our tuning component (OMPICollTune). This component will then select the next algorithm (AlgIDs) for a pending collective call based on a probability distribution. At the start of the tuning process, the selection follows a uniform probability distribution. Therefore, each collective algorithm will have the same chance of being selected. Our tuning component records the start and the end timestamp for each collective call encountered during the application run.[1] Notice that the runtime measurements are taken by the processes while executing an MPI application. Therefore, the runtime of a collective may be impacted by the process' arrival pattern caused by the application's load imbalance. However, this is an advantage of this tuning process, as it measures collective calls in real-world situations. Yet, the current, proof-of-concept tuning approach is sensitive to the actual node allocation given by the scheduler, as load imbalance characteristics may change depending on the set of nodes allotted to a job, which suggests that the performance model could further be extended by including parameters such as the bisection bandwidth or the largest latency between compute nodes.

d) Updating the Performance Model: Now, we can collect performance data for a set of applications, but how much performance data we need before updating the model is another question. Currently, we work with batch sizes of 10 application runs. The performance dataset (performance traces) are then used to update the performance model. The blue boxes (cells) in Figure 1 represent TCs for which performance data has already been measured. For the empty boxes, e.g., the yellow box, we try to predict the performance model using machine learning. To that end, we learn a regression model for each algorithm (AlgID) in order to predict its runtime for a TC.

e) Adapting Selection Probabilities: Our performance model can now be used to update the probability distribution in the algorithm selection process for the next batch of application runs. If we find that one algorithm has shown a particularly poor performance (e.g., a linear Alltoall algorithm), this algorithm will get a very small probability of being selected in the next round. The minimum and the maximum probabilities of selecting an algorithm are parameters of the tuner. The idea is that well performing algorithms will make application runs faster after each model update, as the faster algorithms are more likely to be selected. This training mode (edges 1 to 3 in Figure 2) continues until the probabilities stabilize, or new applications are introduced to the batch scheduler.

IV. Experimental Results

We now present first results of applying our online, iterative tuning strategy on an HPC production system. The results were obtained on the machine *Hydra*, which is composed of 36 compute nodes, where each node comprises two Intel Xeon 6130F @ 2.10 GHz (1152 cores in total). The compute nodes are interconnected via a dual-rail Intel Omni-Path fabric.

A. Online Tuning within Open MPI

The current prototype of our online tuner was developed as a branch of the Open MPI 4.1.x. A link to the Git repository containing our modified version of Open MPI is given in Appendix A.

We would like to share technical details of our implementation. Our tuner within Open MPI is implemented via function callbacks, similarly to the OpenMP tools interface [10]. Thus, we have added function calls to Open MPI, which our tuner can intercept. This way, we can easily disable the tuner by providing empty function callbacks (the overhead of calling an empty function in this context is negligible). We currently support the collectives `MPI_Allreduce`, `MPI_Bcast`, `MPI_Allgather`, and `MPI_Alltoall`, as they are the most frequently used collectives in MPI applications running on supercomputers [1]. Yet, the approach is easily extensible to other collectives and also to other MPI libraries. Listing 1 shows a trimmed version of the Allreduce selection code used by the online tuner. The main purpose of this code is to execute the `algorithm` variant of the Allreduce implementations. Once the online tuner is enabled, we override the variable `algorithm` with our randomly selected algorithm, where the selection probability follows a given probability distribution, which is constantly adjusted.

B. Model Tuning with miniAMR

We now present our results for tuning `MPI_Allreduce` with Open MPI using miniAMR, which is one of the ECP proxy applications [11]. It spends a substantial amount of time in `MPI_Allreduce` [12], [13]. Therefore, selecting an

[1]We use thresholds to limit the number of measurements recorded and the number of processes to participate in the tuning. Thus, the memory overhead is relatively small.

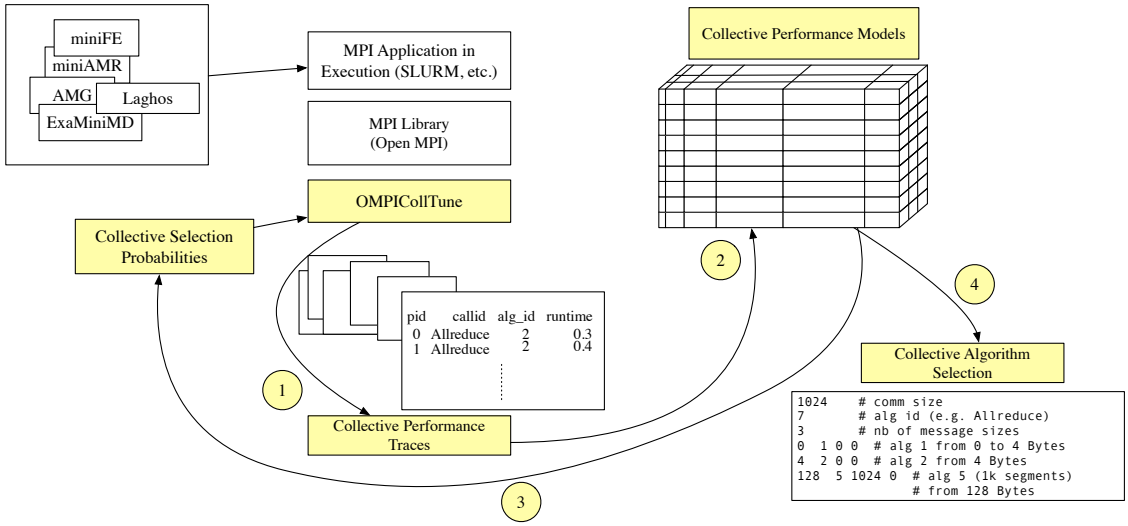


Fig. 2: Autotuning method for MPI collectives. (1) Tracing of MPI applications at runtime. Creating a performance dataset. (2) Injecting performance dataset into performance model. Update of performance model. (3) Recomputation of probabilities to select specific algorithm. (4) Extracting best algorithm for high performance (replacing internal selection strategy).

Listing 1: Pseudocode of our modified selection logic inside Open MPI for `MPI_Allreduce`.

```
if( AT_is_collective_sampling_enabled(MPI_ALLREDUCE) ) {
  // randomly select algorithm (incl. alg configuration)
  our_alg_id = AT_get_allreduce_selection_id(
               bufsize, commsize, operator)

  // translate algorithm and its configuration into OpenMPI
  AT_col_t our_alg = AT_get_allreduce_our_alg(our_alg_id);
  algorithm = our_alg.ompi_alg_id;
  segsize   = our_alg.seg_size;

  AT_record_start_timestamp(MPI_ALLREDUCE, our_alg_id,
                            count * type_size, comm_size);
}
switch (algorithm) {
// ..
case (2):
 res = ompi_coll_base_allreduce_intra_nonoverlapping(..);
 break;
case (3):
 res = ompi_coll_base_allreduce_intra_recursivedoubling(..);
 break;
case (4):
 res = ompi_coll_base_allreduce_intra_ring(..);
 break;
// ..
}
if(AT_is_collective_sampling_enabled(MPI_ALLREDUCE))
  AT_record_end_timestamp(MPI_ALLREDUCE);
```

inefficient Allreduce implementation can be harmful for the performance of this proxy application.

In order to train our prediction model, we run batches of miniAMR, which should reflect a typical batch system, where we would run a set of applications, but only once or twice a day, we would update the performance model. In our experiment, a batch consists of 10 different executions of miniAMR, where the number of compute nodes and the number of processes per node are varied. For example, a batch may contain three runs with 16×32 processes and four runs with 32×1 processes. After executing an entire batch, we update the performance model, which in turn adjusts the selection probability of each algorithm.

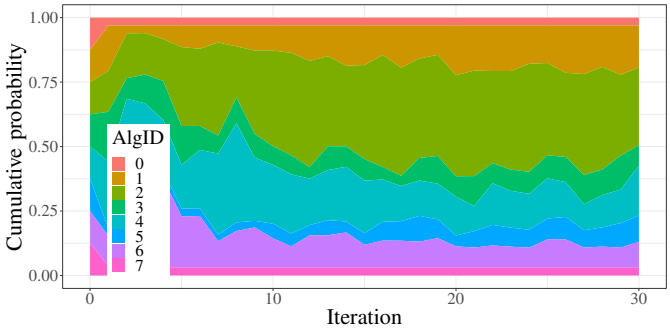
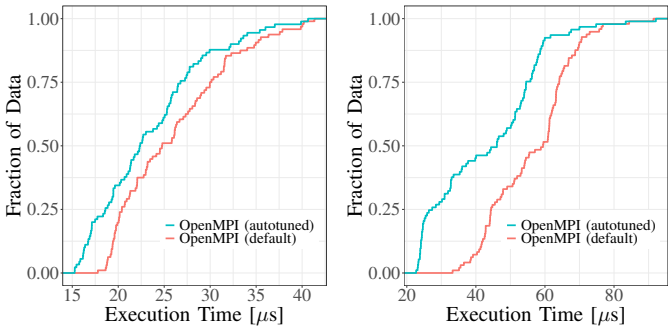


Fig. 3: Progress of selection probabilities of Allreduce algorithms for 32×32 processes and miniAMR.

1) Online Tuning Progress: We show the progress of the selection probabilities for 32×32 processes in Figure 3. At the beginning, at iteration 0, each algorithm has the same chance of being selected. Yet, already after executing the first batch of miniAMR runs, the probabilities have significantly changed. Algorithms 0 and 7 (which represent the *linear* and the *reduce_scatter+allgather* variants) are quickly found to be inferior to the others, and thus, stay at the minimum predefined probability of 3 %. We can also notice that algorithm 2 gets the largest fraction of the probabilities, and thus, will be selected when we determine the best possible algorithms only.

2) Prediction Quality of Performance Model: We like to emphasize that we learn the probabilities, as shown in Figure 3, for each of the *tuning configurations* simultaneously. In the next experiment, we test how well our performance predictions work in practice. Therefore, we predict the performance of each Allreduce algorithm for 24×32 processes. We retrieve a probabilities file from our performance model using several



(a) 8 B (b) 48 B

Fig. 4: ECDF comparison of the running times for 100 calls to `MPI_Allreduce` with 8 B (left) and 48 B for an unseen case of 24×32 processes; *Hydra*.

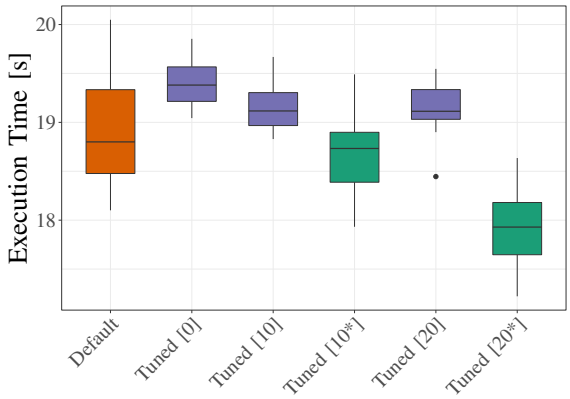


Fig. 5: Overhead and tuning analysis with miniAMR and 32×32 processes; *Hydra*.

regression models (trained using XGBoost [14]) and select the algorithm with the highest probability. This is algorithm 2, which presents the *recursive doubling* variant in Open MPI. We compare the performance of this algorithm to the one that Open MPI selects, which is the *reduce+bcast* version. We show the results of benchmarking runs with ReproMPI [15] for both algorithms, the default of Open MPI and the one found by the online tuner, in Figure 4. We can see that the performance of both algorithms is very similar in this benchmark setting, with 8 and 48 B (which are the message sizes used by miniAMR), although algorithm 2, which the tuner had found, is slightly better. These results show that our tuner can find efficient algorithms purely by measuring the runtime of collectives while executing MPI application.

3) Overhead Analysis: We have yet to answer a very important question: how much overhead does this online tuning method have? It is hard to give a very precise answer, as the overhead depends on so many factors. One of the biggest performance issues is probing a very slow algorithm. Currently, we simply treat all algorithms equally in the beginning. However, the overhead of probing a linear Alltoall algorithm could be significant. Therefore, we could use a blacklist of algorithms for specific cases. Additionally, we can avoid testing different segment sizes when the message size is small, and so forth. Another source of overhead is the number of measurements taken per application run. This number is a parameter of the tuner, and we have taken 10 000 collective runtimes per application execution. If we lower this number, the application runtime will be less affected, but it will take longer to learn a precise prediction model.

Figure 5 presents the overhead analysis and the tuning progress of our approach. In this example, we use 32×32 processes to execute miniAMR. The "Default" version represents the runtime of miniAMR with the default Allreduce selection logic. The individual box plots show the overall runtime for 20 different runs of miniAMR. In addition, the "Tuned" versions represent the runtime of miniAMR using our random selection method. Initially, the online tuner uses a uniform distribution ("Tuned [0]"), and thus, inefficient algorithms are selected as often as effective ones. The median runtime shows an initial overhead of roughly 5 %. After the model is trained, slow algorithms have a smaller selection probability, and thus, the median runtime of miniAMR decreases. The cases with the asterisk (e.g., "Tuned [20*]") denote the runtime distribution if we pick the best algorithm from the model (the algorithm with the highest probability). We observe that the median runtime of miniAMR, after completing 20 batches, is smaller than when using the default logic, which supports our claim that this iterative, online tuning method can indeed be used in practice.

V. Conclusions

Improving the algorithm selection method for MPI collective operations is an effective way for tuning an MPI library on a given target machine. We have shown how to build an algorithm selection model by recording the execution times of collective operations when being executed as part of real application runs. Here, the tuning component randomly selects the next algorithm to be used via a probability distribution, i.e., well-performing algorithms have a higher chance of being selected. Thus, no prior offline micro-benchmarking is required for building a performance model. This alleviates the burden of having to preselect representative message sizes and numbers of compute nodes prior to a tuning process. The performance data is used to update a larger, global performance model that stores runtimes for different node configurations, message sizes, and applications. The larger performance model is iteratively updated, which also adapts the selection probabilities of individual collective algorithms, i.e., faster algorithms will be selected more often in the next round.

We have shown in an experimental case study using the ECP proxy application miniAMR that the collective operation `MPI_Allreduce`, which is heavily used inside this application, can be tuned using our online strategy. In addition, we have shown that our prediction model is able to predict a fast Allreduce algorithm for an unseen node configuration (i.e., number of nodes and processes per node).

Our work on tuning MPI collectives using performance data gathered from application runs is far from complete. In future work, we will tune using a larger set of applications on more compute nodes of larger supercomputers.

References

[1] 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.

[2] 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.

[3] J. Pjesivac-Grbovic, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "Decision trees and MPI collective algorithm selection problem," in *Proceedings of the Euro-Par*, ser. LNCS, vol. 4641. Springer, 2007, pp. 107–117. [Online]. Available: https://doi.org/10.1007/978-3-540-74466-5_13

[4] "Intel® MPI library developer reference for Linux OS," https://www.intel.com/content/dam/develop/external/us/en/documents/mpi-devref-linux.pdf.

[5] 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.

[6] E. Nuriyev, J. Rico-Gallego, and A. L. Lastovetsky, "Model-based selection of optimal MPI broadcast algorithms for multi-core clusters," *J. Parallel Distributed Comput.*, vol. 165, pp. 1–16, 2022.

[7] 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.

[8] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using machine learning," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 259–269.

[9] M. Wilkins, Y. Guo, R. Thakur, P. Dinda, and N. Hardavellas, "ACCLAiM: Advancing the practicality of MPI collective communication autotuning using machine learning," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2022.

[10] A. E. Eichenberger, J. M. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh *et al.*, "OMPT: an OpenMP tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators - 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings*, ser. Lecture Notes in Computer Science, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds., vol. 8122. Springer, 2013, pp. 171–185. [Online]. Available: https://doi.org/10.1007/978-3-642-40698-0_13

[11] "Exascale proxy applications," 2020. [Online]. Available: https://proxyapps.exascaleproject.org/

[12] N. Sultana, M. Rüfenacht, A. Skjellum, P. V. Bangalore, I. Laguna, and K. Mohror, "Understanding the use of message passing interface in exascale proxy applications," *Concurr. Comput. Pract. Exp.*, vol. 33, no. 14, 2021.

[13] B. Klenk and H. Fröning, "An overview of MPI characteristics of exascale proxy applications," in *Proceedings of the 32nd ISC High Performance*, ser. LNCS, J. M. Kunkel, R. Yokota, P. Balaji, and D. E. Keyes, Eds., vol. 10266. Springer, 2017, pp. 217–236. [Online]. Available: https://doi.org/10.1007/978-3-319-58667-0_12

[14] 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.

[15] 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.

Appendix A
Artifact Description

A. Summary of Experiments Reported

We report results of running the ECP proxy application miniAMR. Our experiments are conducted with a modified version of Open MPI (version `4.1.x`).

B. Artifact Availability

The modified version of Open MPI is available at this URL: https://github.com/sebastian-steiner/ompi_pmbs.

The miniAMR application was obtained from: https://github.com/Mantevo/miniAMR.

C. Baseline Experimental Setup, and Modifications Made for the Paper

We have trained miniAMR with the same problem instance for all numbers of processes, which is `-nx 32 -ny 16 -nz 16 -num_tsteps 120`. For each number of processes, we used the decomposition parameters into `npx`, `npy`, and `npz` shown in Table I.

TABLE I: Summary of miniAMR configurations for different numbers of compute nodes N and `ppn`.

N	ppn	npx	npy	npz
4	1	2	2	1
4	32	8	4	4
8	1	4	2	1
8	32	8	8	4
16	1	4	2	2
16	32	8	8	8
32	1	4	4	2
32	32	16	8	8