2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-6460-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPSW63119.2024.00140

PML-MPI: A Pre-Trained ML Framework for Efficient Collective Algorithm Selection in MPI

Mingzhe Han, Goutham Kalikrishna Reddy Kuncham, Ben Michalowicz, Rahul Vaidya,
Mustafa Abduljabbar, Aamir Shafi, Hari Subramoni, Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering
The Ohio State University, Columbus, Ohio, USA
{han.1453, kuncham.2, michalowicz.2, vaidya.64, abduljabbar.1, shafi.16, subramoni.1, panda.2}@osu.edu

***Abstract*—The Message Passing Interface is the de facto standard in high-performance computing (HPC) for inter-process communication. MPI libraries employ numerous algorithms for each collective communication pattern whose behavior is largely affected by the underlying hardware, communication pattern, message size, and number of processes involved. Choosing the "best" algorithm for every possible scenario is a non-trivial task. MPI libraries primarily depend on heuristics for algorithm selection on previously unseen clusters, often resulting in evident slowdowns. Although offline micro-benchmarking tools can exhaustively identify optimal algorithms for all configurations, this is an excessively time-consuming approach. Machine Learning (ML) emerged as an alternate approach. However, most ML-based approaches employ online methods that introduce additional runtime overhead, which makes this impractical at scale. To address this challenge, we propose a pre-trained ML framework that eliminates runtime overhead. Our model requires only a quick inference for each new cluster without necessitating model retraining. It incorporates various hardware features to enhance its adaptability across diverse clusters. Our model's training utilizes tuning data from a broad range of architectures, promoting its versatility and our proposed system exhibits up to 6.3% speedup over default heuristics on systems of up to 1024 cores while significantly minimizing model overhead in comparison to existing methodologies.**

***Index Terms*—MPI, collective communication, auto-tuning, Machine Learning**

I. Introduction

As MPI-based applications continue to scale, communication efficiency has become a top priority. Collective communication operations, which enable efficient data exchange and synchronization among processes, are critical to MPI communication. Most MPI libraries, including popular implementations such as Intel MPI [1], Open MPI [2], and MVAPICH [3], provide multiple algorithms for each collective operation. However, selecting the optimal algorithm for these collective operations can be challenging, as it depends on various factors, including MPI-specific parameters (number of nodes, process-per-node, and message size), hardware features (CPU clock speed, cache size, interconnect and etc.), and network situations. The selection of sub-optimal algorithms can have a substantial impact on performance, leading to an average degradation of up to 35%-45% [4].

This research is supported in part by NSF grants #1818253, #1854828, #2007991, #2018627, #2311830, #2312927, and XRAC grant #NCR-130002.

Several approaches have been proposed to automate this process, including analytical models [5], [6], [7], [8], offline micro-benchmarking [9], [10], and machine learning (ML) approach [4], [11], [12]. Analytical models, which use statistical functions to approximate the runtime of existing algorithms, have shown limited accuracy and present challenges to expanding to new algorithms in production environments. Offline micro-benchmarking, which requires exhaustive pruning of the entire search space, imposes considerable demands on computational resources. Prior studies in machine learning approaches have predominately adopted online training frameworks. Despite attempts to streamline the training process [11], online data collection and subsequent model training following each node allocation can significantly negate the advantages conferred by optimal algorithm selection.

To address the limitations of existing methods, we introduce a novel offline model training framework. This diverges from traditional ML methodologies that solely consider MPI-specific parameters. Contrary to the prevalent belief that the hardware feature space is too complex to explore [11], our framework integrates hardware features into model training. This empowers the ML models to generalize effectively to new clusters and entirely eliminates the necessity of online data collection and model training.

Broadly speaking, there are two categories of collective algorithms: flat and two-level. Two-level collectives introduce additional complexities by distinguishing intra/inter-node communication algorithms. For the purpose of our research, we will focus solely on flat collectives [13]. This focus will streamline our analysis and allow us to effectively evaluate the impact of our machine learning-based approach.

This paper makes the following contributions:

- We have compiled a comprehensive dataset with a wide variety of CPU architectures and interconnects from 18 different clusters.
- We propose a low-overhead ML-based collective auto-tuning framework that can be integrated into MPI libraries. On a system with 8,192 nodes, each featuring 56 cores, our framework can use up to six orders of magnitude fewer core-hours in startup overhead compared to offline micro-benchmarking and the state-of-the-art ML approaches.

979-8-3503-6460-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPSW63119.2024.00140

- We thoroughly study the performance of the proposed framework using an elaborate train-test-split methodology. In terms of collective performance, we demonstrate up to 0.5x - 2x speedups compared to MVAPICH2-2.3.7 and OpenMPI 5.1.0a.

To the best of our knowledge, our proposed design is the first to provide an efficient collective algorithm selection at the lowest MPI application runtime overhead (i.e. in constant time) on unseen clusters and unexplored large-scale node counts.

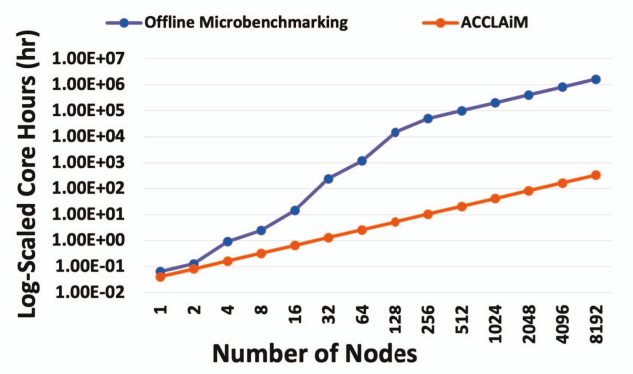


Fig. 1: Core Hours spent by offline micro-benchmarking and ACCLAiM evaluated on TACC Frontera with Intel Xeon Platinum 8280 CPU nodes and Mellanox InfiniBand (EDR) interconnect. Core Hours(hr): number of processes x actual runtime

II. Motivation

Currently, offline micro-benchmarking is the standard methodology for achieving optimal collective algorithm selection [9]. This method generates tuning tables, which are look-up tables mapping particular input parameters to the optimal collective algorithm, enabling fast optimal algorithm selection during application runtime. However, this approach is time-consuming and often takes enormous computing resources to run on a cluster.

Consider a scenario where a user tries to run an MPI-supported application on a new cluster, where no tuning tables are available. Prior to application execution, user must invest substantial time in running the offline micro-benchmarks and generating tuning tables. For application scientists and engineers who operate in a more transient manner, this time-consuming tuning phase is a significant impediment. This challenge is exacerbated in today's HPC landscape, where users frequently migrate their applications across different clusters due to resource availability, cost-effectiveness, or specific hardware requirements.

Machine Learning has offered a faster alternative to offline micro-benchmarking. However, the state-of-the-art ML framework, ACCLAiM [11], requires data collection and model training and inference at application runtime, which poses a significant constraint, particularly for workflows that require frequent job submissions.

Fig. 1 demonstrates the core hours spent for an offline micro-benchmarking tool and ACCLAiM, as the number of nodes evaluated increases. Core hours are the product of the number of processes and actual runtime. We evaluate those frameworks' core hours on TACC Frontera [14] with Intel Xeon Platinum 8280 CPU nodes and Mellanox InfiniBand (EDR) interconnect. MPI_Allgather is chosen for the model overhead evaluation. We calculate the core hours of offline micro-benchmarking with runtime results of our in-house tool that represent such classes. It is important to note that the runtime presented here is derived purely from benchmark results and hence does not include other overhead associated with software implementation. The core hours of ACCLAiM are computed by the model overhead, 5.62 minutes on 128 nodes for MPI_Allgather [11]. Given the limited information, we deliberately ignore the communication overhead, so the actual core hours of ACCLAiM are lower-bounded by the orange line in Fig. 1.

To overcome this significant runtime overhead, we have proposed a pre-trained ML approach by incorporating hardware features, requiring less than a second of model inference overhead during the compilation time. It is important to note that this compilation process is a one-time occurrence for each cluster.

III. Background

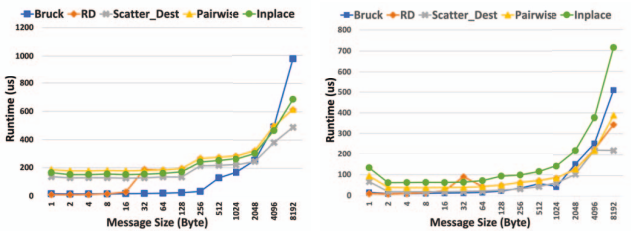
Most MPI libraries offer more than one algorithm for each collective operation. The focus of our research paper revolves around flat algorithms for MPI_Allgather and MPI_Alltoall found in the MVAPICH library. The following paragraphs provide brief descriptions of those algorithms. For more information, readers are encouraged to consult this papers [13].

MPI_Allgather employs a variety of algorithms including Recursive Doubling, Ring, Bruck, and Recursive Doubling Communication algorithms. Recursive Doubling algorithm performs pairwise exchanges between processes, using a recursive halving and doubling approach, resulting in O(log(p)) communication steps. In Ring algorithm, processes are organized in a logical ring structure, and each process iteratively sends its data to its neighboring process until all processes have received the complete gathered data. Bruck algorithm is a simple and efficient algorithm for implementing MPI_Allgather. In each iteration k, process i sends data to process $(i-2^k)$ and receives data from process $(i+2^k)$. Recursive Doubling Communication is a variation of the Recursive Doubling algorithm. Through exchanging subsets of data, this algorithm reduces the amount of data that needs to be sent and received at each communication step, resulting in lower communication overhead compared to the basic Recursive Doubling algorithm.

MPI_Alltoall consists of various algorithms like Bruck, Scatter_Dest (Scatter Destination), Pairwise (Pairwise Exchange), RD (Recursive Doubling), and Inplace (in-place)

Algorithms. The Scatter_Dest Algorithm operates in a scatter-like manner, where each process sends a distinct message to the designated destination process. The Scatter Destination [15] algorithm aims to optimize the distribution of data among processes by minimizing communication overhead and ensuring efficient data exchange. The Pairwise Exchange algorithm requires $p-1$ steps. In each step k, where $1 \leq k < p$, each process determines its target process as $(\text{rank} \oplus k)$ (using XOR operation) and proceeds to exchange data directly with that target process. In the in-place algorithm, the memory usage is optimized by sending and receiving data to the same buffer.

Given an MPI job with a specific number of nodes (#nodes), processes per node (PPN), and message size, which algorithm would be the optimal selection on the current system? Most MPI libraries employ heuristic methods or some form of decision knowledge to choose the "best" algorithm in a given context; however, this is a nontrivial task. As highlighted by Hunold et al., empirical decision trees built on benchmark results form the default tuning strategy of Open MPI 4.0.2, but this can lead to a 30-45% slowdown due to sub-optimal algorithm selection [4]. The challenge lies in the fact that the optimal choice of algorithm heavily depends on the machine's hardware features. Empirical knowledge acquired from one machine cannot be fully transferred to another, even with an identical job size and scale. As illustrated in Fig. 2, the performance of MPI_Alltoall algorithms fluctuates when running on different hardwares, as observed on TACC Frontera and internal cluster, MRI. Frontera has Intel Xeon Platinum 8280 CPU nodes and Mellanox InfiniBand (EDR) interconnect, while MRI is equipped with AMD EPYC 7713 64-Core Processor and Mellanox InfiniBand (HDR) interconnect. Despite implementing identical configurations and runtime parameters, noteworthy variations in their performance exist. For instance, Bruck's algorithm, represented by the blue line, significantly outperforms other algorithms on Frontera for message sizes ranging from 32 to 1024, but its performance degrades on MRI for the same message range. On the contrary, Scatter_Dest's algorithm, depicted as the grey line, has a relatively long runtime for smaller message sizes on Frontera, while it performs exceptionally well for the same range on MRI and even emerges as the optimal algorithm at message size 256 and 512.



(a) MPI_Alltoall on Frontera (b) MPI_Alltoall on MRI

Fig. 2: Runtime comparison between various MPI_Alltoall algorithms on 2 nodes and 16 process-per-node across different clusters

Apart from hardware factors, network congestion can also impact collective algorithm selection. In order to mitigate the impact of the dynamic factors, we gathered the training and testing dataset as in Table I and the performance results by averaging multiple iterations of experiments. Recent approaches in [11], [16] primarily adopt online methods where dynamic factors can be circumvented. However, these online approaches often come with ineligible model overhead. In this work, we acknowledge the noise in our data caused by dynamic factors but posit that by considering static hardware features, we can still improve over the default tuning strategies, which only take into account MPI-specific features.

IV. Design

The proposed framework mainly comprises two stages: offline training and online inference. As shown in Fig. 3, the proposed ML training framework involves collecting cluster-specific and MPI-specific features using a feature extraction script which uses built-in Linux commands to obtain cluster-specific features such as CPU Clock Frequency, L3 Cache, core count, and number of nodes. These features are subsequently employed by the ML model during the inference process. Notably, this ML model is pre-trained and then shipped along with the MPI library, thereby obviating the necessity for end-users to train the model independently.

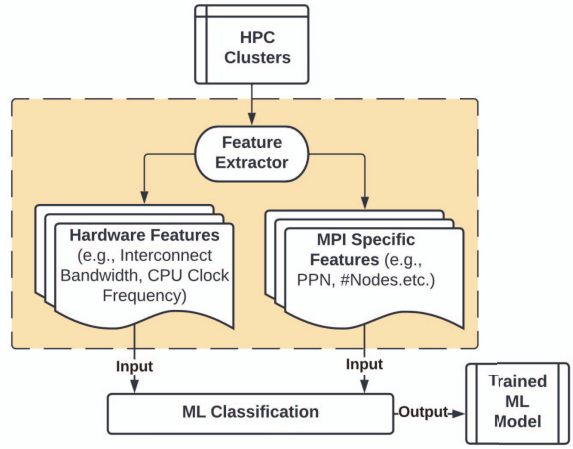


Fig. 3: Offline Training Framework

As depicted in Fig. 4, the framework examines whether a tuning table for the current cluster exists during the MPI library compilation. If such a table is present, the framework bypasses the ML tuning process, opting to use the existing tuning table instead. Conversely, if no such tuning table exists for the cluster, the framework initiates the process of extracting the MPI-specific features and hardware features. The pre-trained ML model from offline training stage subsequently takes both sets of features as input and generates the tuning tables. These tables are stored in a readily accessible JSON format for use during the MPI application runtime.

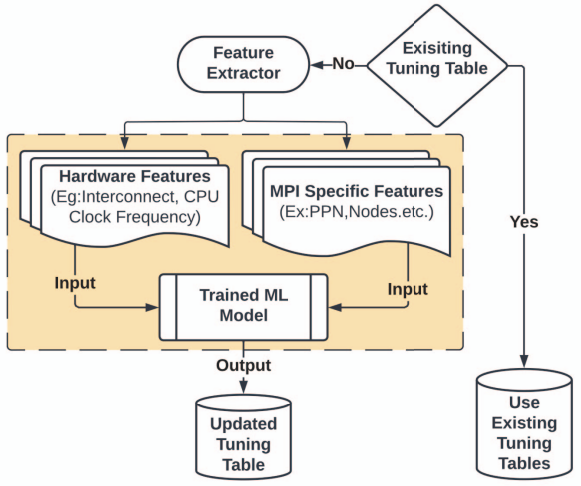


Fig. 4: Online Inference Framework

V. Implementation

A. Hardware Features

Hardware features play a crucial role in collective algorithm selection for efficient performance of MPI collective operations in HPC systems. Bandwidth and latency of the communication network have a significant impact on collective operation performance, favoring different algorithms for different network configurations. The core/thread count, cache size, and memory bandwidth also influence collective algorithm performance. For example, pipelined algorithms that divide data into smaller chunks and process them in a pipelined manner can efficiently overlap communication and computation, leading to improved performance in multicore systems with high memory bandwidth.

Therefore, The hardware features we integrate into our study include CPU maximum clock speed, L3 cache size, memory bandwidth, core-count, thread-count, number of sockets, number of NUMA nodes, number of PCIe lanes, PCIe version, and Host Channel Adapter (HCA)'s link speed and link width. We chose the maximum clock speed over the base clock speed because most processors adapt their clock speeds based on the current workload. Since MPI jobs typically have high workloads, the maximum clock speed is a more accurate feature to simulate MPI runtime environments. The output of the `lscpu` command also reveals another processor-related feature: the number of threads per core. However, we exclude this because it is CPU-dependent and would lead to feature dependency in model training. Instead of using categorical features for the HCA names, we opted for the underlying features: link speed and width.

The current challenge is to determine the most relevant features for each collective. Feature importance is calculated by measuring the decrease in Gini impurity for each feature used to split nodes in the decision trees. This decrease in impurity is accumulated for each feature across all trees in the Random Forest model. The accumulated values are then normalized to obtain the relative importance of each feature. The greater the decrease in impurity, the more important the feature is for making predictions. The feature importance scores for those hardware features and MPI-specific features are displayed in Fig. 5 and Fig. 6. The top 5 features are selected based on this ranking to avoid overfitting issues.

$$I_G(p) = 1 - \sum_{i=1}^{C} p_i^2 \tag{1}$$

$I_G(p)$ is the Gini Impurity, C is the number of classes, and p_i is the probability of class i at the node.

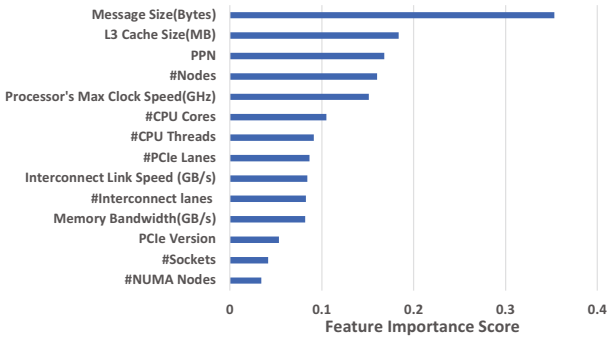


Fig. 5: Feature Importance Score of MPI-Specific Features and Hardware Features Based on Gini Impurity for MPI_Allgather

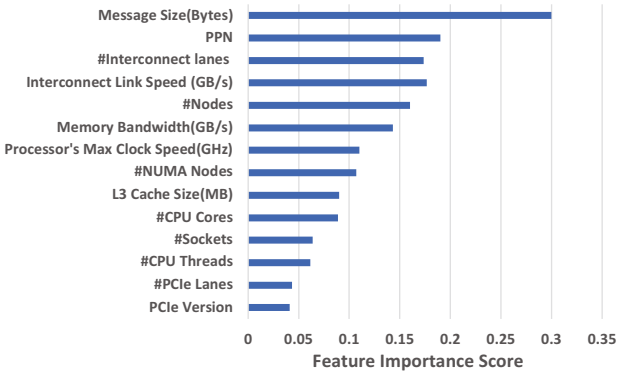


Fig. 6: Feature Importance Score of MPI-Specific Features and Hardware Features Based on Gini Impurity for MPI_Alltoall

Figures 5 and 6 show how MPI-specific features like message size are the dominant parameters for determining the algorithms. However, certain hardware features also play crucial roles, such as L3 cache size for MPI_Allgather and interconnect bandwidth for MPI_Alltoall. The interconnect bandwidth is represented in the figures as interconnect speed and the number of interconnect lanes.

Since MPI_Allgather involves fewer and smaller data transfers between processes, cache size can have a more significant impact on performance. A larger cache size can help reduce

cache misses and the time spent waiting for data to be fetched from memory. On the other hand, MPI_Alltoall involves sending data to all other processes and receiving data from all other processes, resulting in a higher total amount of data exchanged compared to MPI_Allgather. Therefore, interconnect bandwidth becomes more critical for MPI_Alltoall to ensure efficient communication and minimize bottlenecks. These findings highlight the importance of considering MPI parameters and specific hardware features when selecting algorithms for different collective communication patterns.

B. Dataset

Investigating the inherent correlation between hardware features and collective algorithms requires a broad spectrum of distinct clusters into the training data. The list of clusters encompasses RI2, RI, HASWELL, Catalyst, Spock ORNL, ROME, TACC Frontera, LLNL, Frontera RTX partition, ARM Hartree, Mayer, Ray, Sierra, Bridges, Bebop, TACC KNL, TACC Skylake, and MRI. RI2, RI, and MRI are internal clusters. Comprehensive information about this dataset, consisting of over 9000 records for both MPI_Alltoall and MPI_Allgather, can be found in Table I. This dataset covers a wide variety of architectures. It covers many Intel, AMD, and ARM processors and a plethora of InfiniBand and OmniPath interconnects. Our ML model can leverage this dataset to more accurately capture the relationship between hardware features and collective algorithms, leading to better algorithm selection and improved HPC application performance.

C. Model Selection

The choice of the model is the next critical step following the selection of the dataset and hardware features. For the complexity of this dataset (fewer than 10,000 data points) with limited number of features (14), it is common practice to use ML instead of neural networks which incurs underfitting and larger inference overhead. Therefore, we focused on traditional Machine Learning models, specifically Gradient Boosting (GradientBoost), K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Random Forest (RF). Before delving into the specifics of our approach, we first provide an overview of the machine learning models selected for evaluation in this study.

K-nearest Neighbor (KNN) [17] classifies a new instance based on the majority label of its nearest neighbors in the feature space. While KNN is easy to understand and implement, it can suffer from high computational costs in high-dimensional spaces and is sensitive to irrelevant or redundant features.

Support Vector Machine (SVM) [18] is a binary classification algorithm that aims to find the optimal hyperplane that maximizes the margin between the two classes. SVMs are effective in high-dimensional spaces and when the classes are linearly separable. However, they may underperform when the data is noisy or overlaps significantly.

Gradient Boosting [19] is a powerful ensemble method that builds a robust predictive model by combining multiple weak models, typically decision trees. The algorithm iteratively adds new models to the ensemble to correct the errors made by the existing ensemble.

Random Forest [20] is another ensemble learning method that operates by constructing multiple decision trees during training and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. It introduces randomness into the model-building process, which helps to create more diverse trees and reduce overfitting. Random Forests are robust to outliers and can handle unbalanced datasets.

We initiated the experiments with simpler models like KNN and SVM but encountered issues of underfitting. Consequently, we experimented with more complex models like Gradient Boosting and Random Forest. These experiments were CPU-based and made use of the Scikit-learn 1.2.2 package [21]. After conducting extensive hyperparameter tuning for each ML model, we observed that Random Forest outperformed other models in test accuracy for MPI_Alltoall and MPI_Allgather, as shown in Table II. Hence, we selected Random Forest for our framework due to its superior understanding of the dataset.

The algorithm selection task in this work is formulated as a classification problem. One of the major challenges in such a problem setting is the potential class imbalance, which could lead to overfitting issues. To circumvent this, we employ the Area Under the Curve (AUC) metric as an evaluation criterion during the cross-validation stage of model training, as opposed to the conventional accuracy metric. This choice of metric is more robust to class imbalance and offers a more comprehensive view of the model's predictive performance.

D. Accuracy Metrics

The classification accuracy of the ML model is evaluated in three different ways. The dataset mentioned in the previous section was split into training and testing datasets using three different methods: split based on the number of nodes, split based on the cluster names, and random split. The corresponding test accuracy is shown in Table III.

- **Random Test Accuracy**: This is the test accuracy based on the conventional training/testing data split method, which randomly selects 70% of the total dataset as training data and uses the remaining 30% as testing data. This classification accuracy showcases how the model performs as a traditional ML model.
- **Cluster Test Accuracy**: In addition to the above, we split the dataset based on cluster. As shown in Table I, the dataset is comprised of data derived from different clusters. We selected around 70% of the total dataset that only belongs to specific clusters as training data. Since the test data belongs to clusters not exposed to the ML model during the training and validation procedure, we can test the model's adaptability to new, unseen clusters.
- **Node Test Accuracy**: The third test accuracy derives from splitting the data based on the number of nodes, and the model is trained on a smaller number of nodes and tested on a larger number of nodes. This test accuracy demonstrates the scalability of the ML model's tuning

TABLE I: Dataset Overview

Cluster Name	Processor	Interconnect	#nodes	#ppn	#msg size	#samples
RI2	Intel Xeon CPU E5-2680 v4 @ 2.40GHz	Mellanox InfiniBand (EDR)	5	6	21	609
RI	Intel Xeon CPU E5630 @ 2.53GHz	Mellanox InfiniBand (QDR)	1	2	21	42
Haswell	Intel Xeon CPU E5-2687	Mellanox InfiniBand (HDR)	3	6	21	336
Catalyst	FUIJITSU A64FX	Mellanox InfiniBand (EDR)	4	6	21	483
Spock	AMD EPYC 7763 64-Core	Mellanox InfiniBand (HDR)	5	8	21	756
Rome	AMD EPYC 7601 32-Core	Mellanox InfiniBand (EDR)	4	10	21	777
Frontera	Intel Xeon Platinum 8280 CPU @ 2.70GHz	Mellanox InfiniBand (EDR)	5	8	21	756
LLNL	AMD EPYC 7401 48-Core	Mellanox InfiniBand (EDR)	5	6	21	588
Frontera RTX	Intel Xeon CPU E5-2620 v4 @ 2.10GHz	Mellanox InfiniBand (FDR)	5	5	21	504
Hartree	Cavium ThunderX2 CN9975	Mellanox InfiniBand (FDR)	3	5	21	294
Mayer	Cavium ThunderX2 CN9975	Mellanox InfiniBand (EDR)	4	7	21	567
Ray	IBM POWER8 S822LC	Mellanox InfiniBand (EDR)	4	3	21	168
Sierra	IBM POWER9 AC922	Mellanox InfiniBand (EDR)	5	8	21	819
Bridges	Intel Xeon CPU E5-2695 v3 @ 2.30GHz	Intel Omni-Path	5	6	21	567
Bebop	Intel Xeon CPU E5-2695 v4 @ 2.10GHz	Intel Omni-Path	6	5	21	525
TACC KNL	Intel Xeon Phi CPU 7250 @ 1.40GHz	Intel Omni-Path	6	6	21	567
TACC Skylake	Intel Xeon Platinum 8170	Intel Omni-Path	5	8	21	756
MRI	AMD EPYC 7713 64-Core	Mellanox InfiniBand (HDR)	4	8	16	491

TABLE II: Test Accuracy Comparison among GradientBoost, RF, KNN and SVM after Hyperparameter Tuning

Collective	RF	GradientBoost	KNN	SVM
MPI_Allgather	88.8%	80.5%	64.1%	67.3%
MPI_Alltoall	89.9%	78.4%	61.9%	60.4%

TABLE III: Classification Accuracy

Collective	Random Test Accuracy	Cluster Test Accuracy	Node Test Accuracy
MPI_Allgather	88.8%	84.4%	79.8%
MPI_Alltoall	89.9%	82.7%	86.7%

strategies when scaled up to exascale systems. Hence, while the experiments conducted in this paper were limited to 16 compute nodes, we have observed analogous outcomes when implementing the framework on larger scales.

VI. Evaluation Setup

The previous section outlines the design of the ML model and the dataset. This section focuses on the hardware systems, benchmark tools, and applications used for runtime evaluation, comparing our tuning strategy with the baseline framework.

A. Hardware

Our experimental evaluations were performed on two HPC systems: MRI and TACC Frontera. The basic properties of these machines are displayed in Table I. TACC Frontera is ranked 19th on the TOP500 list of Supercomputers located at the University of Texas. MRI is an internal cluster with a larger number of processes per node (PPN), which helps examine the performance of the tuning strategies when dealing with a greater number of processes.

B. Software

We utilize the OSU Micro-Benchmark Suite (OMB) [22], a widely-used tool for MPI performance evaluation, as our benchmark tool. The applications chosen for evaluation are Gromacs and MiniFE.

Gromacs [23] is a versatile package for performing molecular dynamics simulations, primarily designed for biochemical molecules such as proteins, lipids, and nucleic acids. It is widely used in computational chemistry and biophysics to study the dynamic behavior of biomolecules and understand their interactions.

MiniFE [24], or the Mini Finite-Element application, is an HPC proxy application that models the computation and data movement characteristics of unstructured implicit finite element codes. It is a simplified version of a real-world application, providing an easy-to-understand environment for evaluating HPC systems and optimization strategies.

VII. Experiment Results

This section delves into the discussion of the evaluation results on the proposed framework, PML-MPI. We use the OSU-micro-benchmark tools and conduct experiments on two systems: Frontera and MRI, as introduced in Section VI. The ultimate goal of this research is to accelerate applications through a better selection of collective algorithms. Thus, we evaluate the actual runtime performance. In this section, we evaluate the performance of an open-source MPI Library, MVAPICH2 2.3.7 [3] developed by the Ohio State University with ML-based tuning strategies on both micro-benchmark and application levels.

A. Addressing the Overhead in Previous Work

Fig. 7 illustrates the core hours expended for offline micro-benchmarking, ACCLAiM, and our proposed design, as the number of evaluated nodes expands. We assess these frameworks' core hours on the TACC Frontera system, which employs Intel Xeon Platinum 8280 CPU nodes and Mellanox InfiniBand (EDR) interconnect. MPI_Allgather is selected for the model overhead evaluation. The core hours of the proposed design remain nearly constant, primarily because it only involves the ML model inference time, which only

requires one process. Thus, scaling up to a larger number of nodes does not noticeably increase this inference time. As depicted in Fig. 7, compared to offline micro-benchmarking at 32 nodes, our proposed design exhibits a speedup of 10E+06 in model overhead. When juxtaposed with ACCLAiM's runtime model overhead at 128 nodes, our design delivers a speedup of 10E+04. While our proposed framework can potentially demonstrate a much more significant speedup in overhead for larger system sizes (such as 8k nodes), we confine the discussion to a comparison with offline micro-benchmarking at 32 nodes and ACCLAiM at 128 nodes. This approach ensures objectivity, as these points of comparison are directly calculated or derived from published results [11].

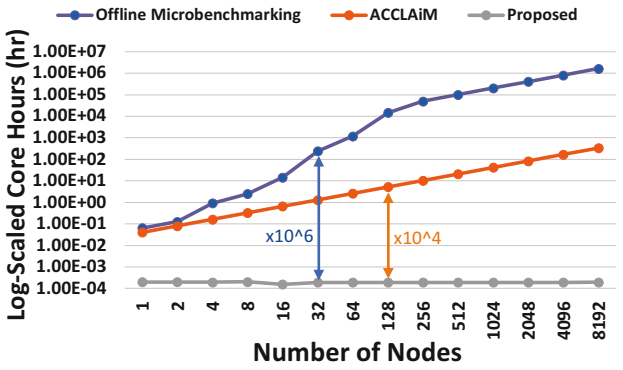


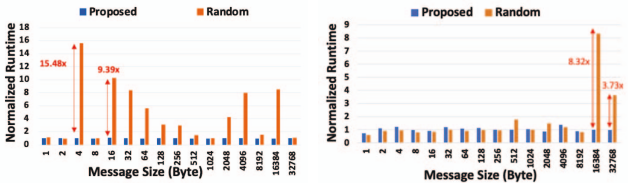
Fig. 7: Core Hours spent by offline micro-benchmarking, ACCLAiM and the Proposed Framework evaluated on TACC Frontera with Intel Xeon Platinum 8280 CPU nodes and Mellanox InfiniBand (EDR) interconnect. Core Hours(hr): number of processes x actual runtime

B. Model Applicability

The ML model is redundant if the random selection of algorithms also leads to comparable runtime performance. Fig. 8a and 8b have demonstrated that the proposed framework is compared with random algorithm selection on Frontera for #node=16 and PPN=56. Choosing the algorithms randomly has caused substantial slowdowns for MPI_Allgather and MPI_Alltoall on different message sizes. For the MPI_Allgather operation, our framework demonstrates a speedup of 15.48x and 9.39x, while for the MPI_Alltoall operation, we observe a speedup of 8.32x and 3.73x on large message sizes. The overall performance comparison between the proposed framework and random selection can be found in the next section. Similar trends are observable across other configurations. Therefore, a strategy relying solely on random algorithm selection is insufficient for real-world use cases and a robust decision model is crucial to optimize MPI performance.

C. Cluster-Based Benchmark Results

The ML model is trained utilizing the dataset delineated in Section V, with any data associated with Frontera and



(a) #node=16, PPN=56 on MPI_Allgather

(b) #node=16, PPN=56 on MPI_Alltoall

Fig. 8: Normalized Runtime Comparison Between the Proposed and Random Selection of Algorithms for MVAPICH2 2.3.7 on TACC Frontera

MRI expressly excluded during the respective experiments. This methodology enables us to evaluate the model's runtime performance on previously unencountered clusters.

As shown in Fig. 9, our proposed framework demonstrates a clear performance advantage over MVAPICH2 2.3.7-default for certain message sizes on Frontera. For instance, in Fig.6.(b), when message sizes are 4096 and 8192 bytes, our framework selects faster algorithms for MPI_Alltoall, showing 36.6% and 36.33% speedup. A similar trend is observed for MPI_Allgather on Frontera, with runtime advantages at message sizes of 4 and 2048 bytes, achieving 59.97% and 44.29% speedup. It's critical to highlight that when both frameworks choose the same algorithms, any variances in runtime are primarily attributable to network conditions. For message sizes of 4096 and 8192, our method tends to select more efficient algorithms, reflecting the sensitivity of collective algorithms to message size, as illustrated by Fig. 9. These findings underscore our approach's enhanced ability to adapt algorithm selection in response to changes in message size.

One may wonder if this is purely a serendipitous occurrence. The same pattern is also observed for the MRI cluster, as shown in Fig. 10. For both MPI_Alltoall and MPI_Allgather, our proposed framework selects better algorithms as in Fig. 10. (c) and . (d), where 150.11% and 154.46% speedup is observed as it considers the hardware features of the current hardware system, whereas MVAPICH2 2.3.7 default tuning tables rely on a static tuning table, which lacks optimization for the specific cluster.

For Frontera, we tested various configurations, including 1, 2, 4, 8, and 16 nodes, with process-per-node (PPN) settings of 28 (half-subscription) and 56 (full-subscription). For MRI, we tested various configurations, including 1, 2, 4, and 8 nodes, with process-per-node (PPN) settings of 64 (half-subscription) and 128 (full-subscription).

Compared to the exhaustive offline micro-benchmarking of the optimal algorithms, our proposed method still shows an acceptable slowdown. On Frontera, the proposed method has a 0.6% slowdown for MPI_Allgather and a 5.6% slowdown for MPI_Alltoall. On MRI, the proposed method shows a 5.1% slowdown for MPI_Allgather and 5.8% for MPI_Alltoall. The slowdown of the ML-based approach is bounded by 6%, and our approach significantly curtails the time expended on

running offline micro-benchmarking.

These diverse configurations allow us to assess the performance of our tuning strategies across different levels of system utilization and concurrency, providing valuable insights into the effectiveness of our machine learning-based approach in optimizing application performance.

On MRI, the average speedup of the proposed approach for MPI_Allgather and MPI_Alltoall is 6.3% and 2.5%, respectively, while the average speedup compared to the random tuning strategy for MPI_Allgather and MPI_Alltoall is 2.96x and 2.76x respectively.

To provide a more exhaustive evaluation of the model's performance on unencountered clusters, we also juxtapose the performance of our proposed method with that of Open MPI 5.1.0a. (OMPI). As shown in Fig. 11, our proposed approach shows speedup for larger message sizes, particularly beyond 4k, where 49.12% and 57.67% speedup can be found for MPI_Alltoall , and 54.01% and 36.22% speedup for MPI_Allgather. Fig. 11 .(a) and .(b) indicate a slight slowdown with our method when the message size is 1, which is attributable not to suboptimal algorithm selection but to prevailing network conditions.

D. Node-Based Benchmark Results

Following the evaluation of the model's adaptability to unencountered clusters, the subsequent metric to consider is the model's scalability on larger clusters. We conduct this evaluation by training the model with the dataset, specifically excluding any data involving a larger number of nodes.

On Frontera, we train the ML model with #nodes=1,2,4,8, and the runtime performance is compared with the MVAPICH2 2.3.7 default for #nodes=16. Fig. 12c/12d shows 13.2% and 43.45% improvement with 2048 and 4096-byte messages, respectively. On MRI, we train the ML model with #nodes=1,2,4, and the performance is assessed at #nodes=8, as shown in Fig. 12a and 12b, we can see 74.07% speedup at message size 1024 bytes for MPI_Allgather and 58.55% and 49.63% speedup at message 16384 and 32768 bytes for MPI_Alltoall.

E. Application Results

Since the model's adaptability and scalability have been established in the above sections, we employ Gromacs and MiniFE to investigate the actual speedup within production environments.

We compile Gromacs on the Frontera with the configuration flags "-DGMX_MPI=on" and "-DGMX_BUILD_OWN_FFTW=ON." As of Fig. 13, the number of processes is denoted as #Processes. We use the BenchMEM benchmark [25] to gather runtime data. For strong scaling, the runtime diminishes as we increase the number of processes. Scalability is forfeited when scaling up to approximately 224 processes.

As shown from Fig. 13, our proposed framework demonstrates performance benefits compared to the MVAPICH2 2.3.7 default for both Gromacs and MiniFE. The speedups are consistent with the OMB benchmark results.

For Gromacs, our design achieves a speedup of 2.90% compared to the default tuning strategy and a speedup of 19.39% compared to the random selection. For MiniFE, our design's speedup compared to the default tuning strategy is 4.43%, and the speedup compared to the random selection is 20.66%.

VIII. Related Work

The optimization of MPI collectives has been a subject of extensive research, dating back over two decades [26], [27]. This field has become even more active in the past ten years as MPI collective operations have grown to meet the demands of HPC and Deep Learning applications [28], [29].

Several efforts have been made regarding analytical models. Pjesivac-Grbovic et al. [5] extend accepted point-to-point communication models like Hockney, LogP/LogGP, and PLogP to collective operations. They compare predictions from these models against experimental data and use the results to construct an optimal decision function for the broadcast collective.

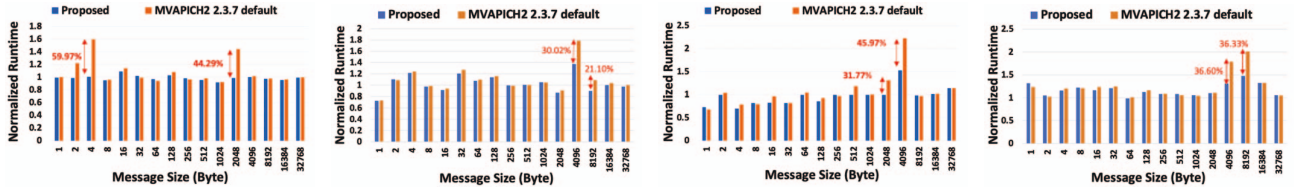
Nuriyev et al. [6] furthered this by developing analytical models derived from code implementation in place of abstract mathematical definitions. they separately estimated model parameters such as latency and bandwidth for each collective algorithm, integrating them into the corresponding communication experiment.

The most recent progress made with analytical models is HAN [7]. HAN uses homogeneous collective communication modules as submodules for each hardware level and treats them as tasks. These tasks are then organized to perform efficient hierarchical collective operations. HAN's design enables easy substitution of submodules to adapt to new hardware, offering a resilient and flexible solution for contemporary platforms while future-proofing upcoming HPC systems.

OMPICollTune [16] employs an online probability model which incorporates the testing of different algorithms directly into the MPI library. Recorded performance results are used to update this model, with the tuner adjusting the probabilities of selecting an algorithm so that slower algorithms are less likely to be chosen.

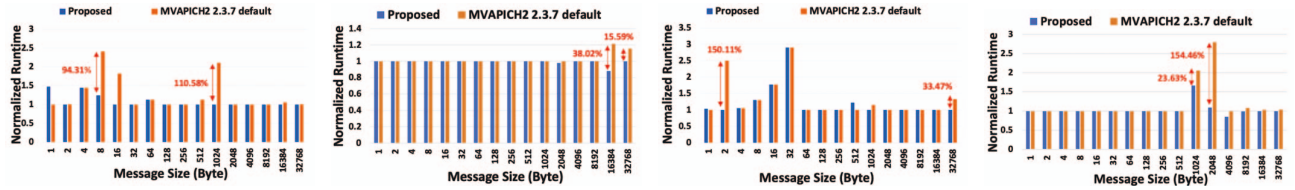
IX. Conclusion

In this work, we develop an ML tuning framework on top of a comprehensive dataset spanning a broad array of architectural variations and introduce a pre-trained ML model for the optimal selection of MPI collective algorithms. This method significantly minimizes model overhead while enhancing performance compared to default heuristics and state-of-the-art techniques. The proposed system exhibits a speedup of up to 6.3% over default heuristics on systems with up to 16 nodes. While this study is performed against the MVAPICH2-2.3.7 software, it is applicable to any MPI library. This work provides an alternative solution to the challenges of incorporating hardware features in machine learning models and



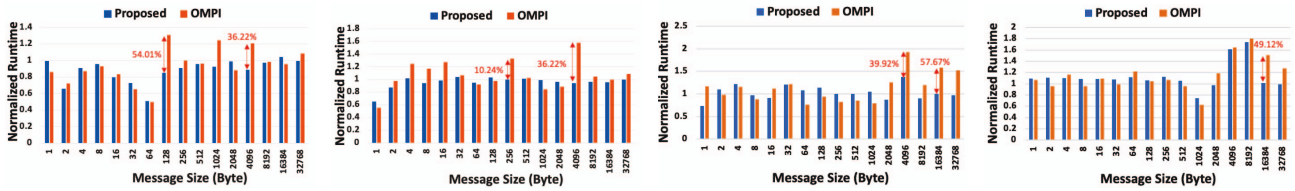
(a) MPI_Allgather, #nodes=16, PPN=56 (b) MPI_Alltoall, #nodes=16, PPN=56 (c) MPI_Allgather, #nodes=16, PPN=28 (d) MPI_Alltoall, #nodes=16, PPN=28

Fig. 9: Comparison of the algorithm selection strategies between the Proposed design and MVAPICH2 2.3.7 default on TACC Frontera using Cluster-Based Benchmarks



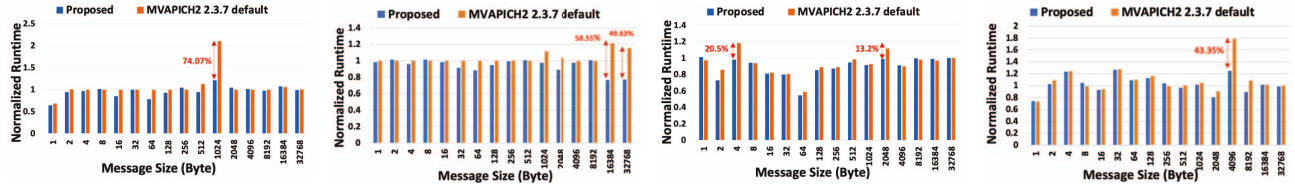
(a) MPI_Allgather, #nodes=8, PPN=128 (b) MPI_Alltoall, #nodes=8, PPN=128 (c) MPI_Allgather, #nodes=8, PPN=64 (d) MPI_Alltoall, #nodes=8, PPN=64

Fig. 10: Comparison of the algorithm selection strategies between the Proposed design and MVAPICH2 2.3.7 default on MRI using Cluster-Based Benchmark Results



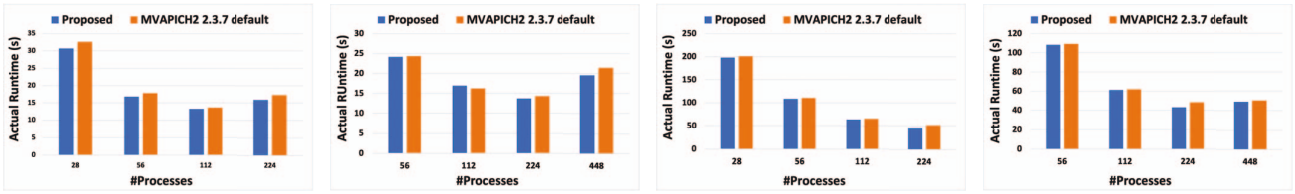
(a) MPI_Allgather, #nodes=16 (b) MPI_Allgather, #nodes=8 (c) MPI_Alltoall, #nodes=16 (d) MPI_Alltoall, #nodes=8

Fig. 11: Comparison of the algorithm selection strategies between the Proposed design and Open MPI 5.1.0a at PPN=56 (full-subscription) on TACC Frontera using Cluster-Based Benchmark Results



(a) MPI_Allgather, MRI, #nodes=8 (b) MPI_Alltoall, MRI, #nodes=8 (c) MPI_Allgather, Frontera, #nodes=16 (d) MPI_Alltoall, Frontera, #nodes=16

Fig. 12: Comparison of the algorithm selection strategies between the Proposed design and MVAPICH2 2.3.7 default at PPN=56 (full-subscription) using Node-Based Benchmark Results



(a) MiniFE, PPN=28 (b) MiniFE, PPN=56 (c) Gromacs, PPN=28 (d) Gromacs, PPN=56

Fig. 13: Actual Runtime Comparison between the Proposed design and MVAPICH2 2.3.7 default on MiniFE and Gromacs

sets a new standard for optimizing MPI collective algorithm selection.

In future work, we intend to build upon our current research on MPI_Alltoall and MPI_Allgather by advancing the ML framework for selecting collective algorithms across a broader range of MPI collective communication, especially those with more intricate communication hierarchies.

REFERENCES

[1] [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html

[2] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 2004, pp. 97–104.

[3] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, vol. 52, p. 101208, 2021.

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

[5] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, pp. 127–143, 2007.

[6] E. Nuriyev and A. Lastovetsky, "Efficient and accurate selection of optimal collective communication algorithms using analytical performance modeling," *IEEE Access*, vol. 9, pp. 109 355–109 373, 2021.

[7] X. Luo, W. Wu, G. Bosilca, Y. Pei, Q. Cao, T. Patinyasakdikul, D. Zhong, and J. Dongarra, "Han: a hierarchical autotuned collective communication framework," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 23–34.

[8] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: self tuned adaptive routines for MPI collective operations," in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 199–208.

[9] M. Chaarawi, J. M. Squyres, E. Gabriel, and S. Feki, "A tool for optimizing runtime parameters of Open MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, September 7-10, 2008. Proceedings 15*. Springer, 2008, pp. 210–217.

[10] "Intel MPI library developer reference for linux os," Available at https://www.intel.com/content/www/us/en/content-details/740630/intel-mpi-library-developer-reference-for-linux-os.html (2022/08/26).

[11] M. Wilkins, Y. Guo, R. Thakur, P. Dinda, and N. Hardavellas, "ACCLAiM: Advancing the practicality of mpi collective communication autotuning using machine learning," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 161–171.

[12] M. Wilkins, Y. Guo, R. Thakur, N. Hardavellas, P. Dinda, and M. Si, "A FACT-based approach: Making machine learning collective autotuning feasible on exascale systems," in *2021 Workshop on Exascale MPI (ExaMPI)*. IEEE, 2021, pp. 36–45.

[13] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[14] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The evolution of leadership computing at the national science foundation," in *Practice and Experience in Advanced Research Computing*, 2020, pp. 106–111.

[15] R. Kumar, A. Mamidala, and D. K. Panda, "Scaling alltoall collective on multi-core systems," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

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

[17] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[18] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[19] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[20] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1. IEEE, 1995, pp. 278–282.

[21] [Online]. Available: https://scikit-learn.org/stable/whats_new/v1.2.html

[22] "OSU Micro Benchmarks," Available at https://mvapich.cse.ohio-state.edu/benchmarks (2023/06/23).

[23] M. Abraham, A. Alekseenko, C. Bergh, C. Blau, E. Briand, M. Doijade, S. Fleischmann, V. Gapsys, G. Garg, S. Gorelov, G. Gouaillardet, A. Gray, M. E. Irrgang, F. Jalalypour, J. Jordan, C. Junghans, P. Kanduri, S. Keller, C. Kutzner, J. A. Lemkul, M. Lundborg, P. Merz, V. Miletić, D. Morozov, S. Páll, R. Schulz, M. Shirts, A. Shvetsov, B. Soproni, D. van der Spoel, P. Turner, C. Uphoff, A. Villa, S. Wingbermühle, A. Zhmurov, P. Bauer, B. Hess, and E. Lindahl, "Gromacs 2023.1 manual," Apr. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7852189

[24] "Mini finite element application (miniFE)," Available at https://github.com/Mantevo/miniFE.

[25] Kutzner, Páll, Hess, d. Groot, Bock, and Matthes, "A free GROMACS benchmark set." [Online]. Available: https://www.mpinat.mpg.de/grubmueller/bench

[26] S. Sistare, R. Vandevaart, and E. Loh, "Optimization of MPI collectives on clusters of large-scale smp's," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999, pp. 23–es.

[27] L. P. Huse, "MPI optimization for smp based clusters interconnected with sci," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 7th European PVM/MPI Users' Group Meeting Balatonfüred, Hungary, September 10–13, 2000 Proceedings 7*. Springer, 2000, pp. 56–63.

[28] J. Jose, K. Hamidouche, J. Zhang, A. Venkatesh, and D. K. Panda, "Optimizing collective communication in UPC," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 361–370.

[29] S. Li, T. Hoefler, and M. Snir, "NUMA-aware shared-memory collective communication for MPI," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, 2013, pp. 85–96.