Machine Learning Driven Auto-tuning for Non-uniform All-to-All Collectives

Kunting Qi[1], Ke Fan[2], Jens Domke[3], Seydou Ba[3], Venkatram Vishwanath[4], Michael E. Papka[1, 4], and Sidharth Kumar[1]
[1]University of Illinois Chicago, Chicago, IL, USA
[2]Temple University, Philadelphia, PA, USA
[3]RIKEN Center for Computational Science, Kobe, Japan
[4]Argonne National Laboratory, Lemont, IL, USA
{kqi, papka, sidharth}@uic.edu, ke.fan@temple.edu, {jens.domke, seydou.ba}@riken.jp, {venkat, papka}@anl.gov

***Abstract*—Non-uniform all-to-all communication patterns present optimization challenges in parallel computing due to their irregular data distribution and dynamic behavior. While `MPI_Alltoallv` provides the standard interface for such exchanges, achieving optimal performance requires careful selection among multiple implementation variants and tuning of algorithm-specific parameters. This paper presents a data-driven autotuning framework that combines machine learning–based runtime prediction with a lookup-table mechanism for fast configuration selection. The ML model estimates the communication time of each algorithm configuration under a given system setup, allowing the framework to identify the optimal implementation and parameter set based on predicted performance. We validate our approach through comprehensive benchmarking of `MPI_Alltoallv` and two specialized algorithms across varying process counts, message sizes, and tunable parameters. Applied to a real MPI-based transitive closure application on the Fugaku supercomputer, our framework achieves up to 6.03× reduction in communication time over the vendor implementation, providing a detailed understanding of non-uniform collective communication behavior and a practical framework for automatic performance optimization in HPC applications.**

***Index Terms*—MPI_Alltoallv, autotuning, collective communication, machine learning, sensitivity analysis**

I. Introduction

Collective communication primitives are essential to high-performance computing (HPC), enabling scalable data exchange and synchronization among processes. Among these, *non-uniform* all-to-all data exchange (`MPI_Alltoallv`) [1], [2] plays a critical role in numerous real-world applications, including graph analytics [3], sparse matrix computations [4], and scientific simulations [5], where processes exhibit heterogeneous communication patterns with varying message sizes. State-of-the-art *non-uniform* all-to-all implementations ranging from the classic algorithms provided by MPI libraries to specialized methods such as parameterized all-to-all (*ParAta*) and hierarchical all-to-all (*HieAta*), proposed by our prior research [6] [2]. Each implementation exposes configurable tuning parameters (e.g., *radix*, *batch_size*) whose values can dramatically affect performance. Consequently, optimizing non-uniform all-to-all operations involves not only choosing the optimal algorithm but also selecting appropriate parameter configurations. However, this optimization is challenging due to the high variability in message sizes and communication patterns across processes and over time, coupled with the expansive combined space of implementation choices and tuning parameters. Exhaustive search or manual tuning during execution is impractical, and any effective autotuning strategy must incur minimal overhead to ensure net performance benefits. Existing MPI libraries typically rely on fixed heuristics or static decision rules, which cannot adapt effectively to the irregular and dynamic nature of real-world HPC communication workloads. As a result, this necessitates flexible, adaptive tuning frameworks that are capable of autonomously identifying both the optimal implementation and parameters for *non-uniform* all-to-all exchange.

Prior works on optimization for collective routines employ a range of approaches, including analytical modeling [7], [8], empirical benchmarking [9], [8], and machine learning-based autotuning, both offline and online [10], [11], [12], [13], [14], [15], [16]. Efficiently tuning *non-uniform* all-to-all data exchange remains challenging due to the irregularity of message sizes and diverse communication patterns. While prior studies [14], [17], [6] have begun to explore *non-uniform* collectives (such as reduce-scatter), comprehensive analyses addressing the impact of *variable* block sizes on performance and algorithm selection are still lacking. Our work bridges this gap by targeting non-uniform all-to-all (all-to-allv).

The main contribution of this work is an *auto-tuning framework targeting all-to-allv data exchanges* that combines data benchmarking, feature reduction, machine learning (ML) prediction, and lightweight runtime tuning to automatically select the most efficient algorithm and parameters for different communication patterns. The framework is built on top of the *ParAta* and *HieAta* algorithms, which offer parameterized implementations of non-uniform all-to-all data exchanges by exposing parameters such as $radix$ and $batch\_size$. Our work is implemented through the following four main components that together enable efficient, on-the-fly optimization for non-uniform all-to-all communication:

1) *Sensitivity-guided feature reduction:* We conduct a sensitivity analysis to identify the minimal yet most effective set of predictive features, thereby reducing both model complexity and runtime overhead.

2) *ML-based performance modeling:* We develop a predictive model that estimates the runtime of non-uniform all-to-all operations under various algorithms and parameter settings, utilizing large-scale benchmark data.
3) *Efficient offline lookup construction:* We generate a compact lookup table that maps observable runtime features (e.g., process count, average message size, and topology) to the optimal algorithm and configuration.
4) *Lightweight runtime integration:* We implement a C++ runtime wrapper that queries the lookup table to select and invoke the best algorithm at each invocation, adding less than 1% overhead while achieving substantial performance gains on real HPC applications.

Our approach enables dynamic optimization for applications that repeatedly invoke *non-uniform* all-to-all operations with time-varying workloads, automatically adapting both implementation selection and parameter configuration to match evolving communication patterns. This capability is essential for applications exhibiting temporal variations in data distribution and communication requirements. We validate our methodology through comprehensive evaluation using both synthetic benchmarks and production-scale applications on two leadership-class supercomputers: Polaris at Argonne Leadership Computing Facility and Fugaku at RIKEN. Our framework exhibits a performance improvement of $6.03\times$ in communication time over the vendor implementation of *MPI_Alltoallv* when applied to an actual MPI-based graph mining application on the Polaris supercomputer.

II. Background

In this section, we introduce the basics of *non-uniform* all-to-all data exchange, necessary for understanding the subsequent sections. We present three key components: first, we present the formal definition of non-uniform all-to-all (Section II-A), second, we describe two fundamental algorithmic approaches for implementing all-to-all (Section II-B), and finally, we discuss two tunable implementations of *non-uniform* all-to-all that this work is based on (Section II-C).

A. Definition of Non-uniform All-to-all

With P processes, *non-uniform* all-to-all data exchange can be expressed as follows. Every process has a *send buffer* (initialized with data), logically made out of P data-blocks ($S[0 \dots P-1]$), each with an arbitrary number (n) of elements with a certain data-type (e.g., integer or float). Similarly, processes also have a *receive buffer* (initially empty), logically made out of P data-blocks ($R[0 \dots P-1]$). When implemented by `MPI_Alltoallv`, both the send and receive buffers are contiguous 1-D arrays of size $P \times n$ elements where all data-blocks $S[0 \dots P-1]$ and $R[0 \dots P-1]$ are laid out in increasing block order. During communication, a process with rank j $(0 \leqq j \leqq P-1)$ transmits the data-block $S[i]$ $(0 \leqq i \leqq P-1)$ to a process with rank i and receives a data-block from rank i into the data-block $R[i]$, except the data destined for itself.

B. Standard Implementations of All-to-all

Multiple algorithmic approaches have been developed to implement *non-uniform* all-to-all data exchange. These algorithms can be categorized by the value of the parameter *radix*: (1) *radix-2* and (2) *radix-P* approaches, where P denotes the total number of processes. These two categories exhibit different computational complexities: *radix-2* algorithms exhibit logarithmic complexity, completing the exchange process in $log_2(P)$ communication rounds, with each round exchanging multiple data-blocks per process. In contrast, *radix-P* algorithms demonstrate linear complexity, requiring P communication rounds, with each round exchanging one data-block per process. The Bruck algorithm [18] and the Spread-out algorithm [19] are two classic representatives of these two categories, respectively.

Spread-out algorithm: This algorithm is the most intuitive implementation of all-to-all, utilizing *non-blocking* point-to-point communication primitives. In this algorithm, each process first posts all receive requests using `MPI_Irecv` in a loop, then posts all send requests using `MPI_Isend` in a subsequent loop, followed by an `MPI_Waitall` operation to ensure completion of communication. Notably, this algorithm enables each process to communicate with different destination processes per round to avoid potential network congestion.

Bruck algorithm: This algorithm is a classical logarithmic all-to-all algorithm with radix 2. Its efficiency lies in transmitting larger volumes of data with fewer communication rounds. In its original form, it comprises three phases: a local initial data rotation phase, a communication phase with $\log_2 P$ rounds, and a local inverse data rotation phase. The communication phase is characterized by the binary representation of data-block indices, with a varying number of data-blocks exchanged per round. It is a store-and-forward algorithm, where data blocks received in one round are forwarded in subsequent rounds for further transmission.

C. Tunable Implementations of All-to-all

Existing implementations in popular libraries (e.g., MPICH [20] and OpenMPI [21]) primarily focus on two extreme radix values (2 and P), thereby overlooking a substantial, unexplored parameter space between them. The capacity to dynamically tune the algorithm's radix between 2 and P would enable flexible adjustment of communication rounds and data exchange volumes, facilitating fine-grained performance optimization [2]. This paper examines two *non-uniform* all-to-all algorithms introduced in our previous work: parameterized all-to-all (*ParAta*) and hierarchical all-to-all (*HieAta*). Both algorithms incorporate a configurable *radix* parameter that enables smooth transitions between logarithmic and linear algorithmic regimes. Additionally, *HieAta* introduces a second tunable parameter, *batch_size*, which constrains the maximum number of simultaneous communication requests in the network to further mitigate network congestion. These highly tunable algorithms are the primary focus of this paper.

TABLE I
INPUT PARAMETERS FOR ML MODELS

Symbol	Description	Type
P	Total number of processes	Execution
PPN	Number of processes per node	Execution
τ	Network topology	Platform-specific
A	Algorithm selection	Model-specific
r	Radix of algorithm	Algorithmic-specific
b	batch_size: max number of comm requests	Algorithmic-specific
μ_B	Average of data-block sizes	Statistical
$B_{\max}$	Maximum of data-block sizes	Statistical
σ_B^2	Variance of data-block sizes	Statistical
$Q_1(B)$	Sample quantile of data-block sizes (25%)	Statistical
$Q_3(B)$	Sample quantile of data-block sizes (75%)	Statistical
γ_B	Skewness of data-block sizes	Statistical
κ_B	Kurtosis of data-block sizes	Statistical

Parameterized all-to-all (*ParAta*) algorithm: This algorithm is built upon three key ideas to enable the performance tunability for *non-uniform* all-to-all: (1) a logarithmic-time generalized implementation of the Bruck-style algorithm with varying radices, (2) a two-phase data exchange mechanism in each communication round, comprising a metadata exchange followed by actual data transfer, and (3) a strategically sized temporary buffer (T) to support intermediate data exchanges during logarithmic communication rounds. These design ideas enable users to optimize the trade-off between the number of communication rounds and the size of data exchanges, leading to improved performance scalability.

Hierarchical all-to-all (*HieAta*) algorithm: This algorithm leverages a multi-layer hierarchical architecture on modern HPC systems through a decoupled communication structure. This structure separates data exchanges into two distinct phases: intra-node communication, utilizing shared memory within nodes, and inter-node communication, facilitating message transfer across the network. Based on the observation that intra-node communication is dominated by latency while inter-node communication is dominated by bandwidth, intra-node communication employs the *ParAta* algorithm. In contrast, inter-node communication uses the *scattered* algorithm. The scattered algorithm is an optimized version of the spread-out algorithm, which integrates an additional tunable parameter, *batch_size*, to control the maximum number of simultaneous communication requests existing in the network. Consequently, this algorithm incorporates dual parametric controls, with separate tunable parameters governing the intra-node and inter-node communication phases.

III. END-TO-END AUTOTUNING FRAMEWORK

Our work aims to develop an adaptive autotuning framework capable of dynamically selecting both the most efficient all-to-allv implementation and its corresponding optimal parameter configuration. This framework addresses the needs of applications with temporally varying communication patterns, enabling automatic runtime adaptation to achieve optimal performance across evolving workload characteristics.

Our framework operates through two interconnected phases that collectively enable real-time optimization of all-to-allv communication. The first phase (Section IV) conducts sensitivity analysis to identify the subset of parameters with the strongest performance impact, while the second phase develops a machine learning-driven performance model that powers a lightweight lookup mechanism for runtime optimization. Figure 1 illustrates the complete framework architecture. Phase 1 involves exhaustive performance characterization across all feasible parameter combinations and workload distributions, generating a comprehensive dataset of execution behaviors. This empirical analysis is complemented by statistical sensitivity analysis to identify the most influential parameters, thereby effectively reducing the dimensionality of the optimization space while preserving prediction accuracy.

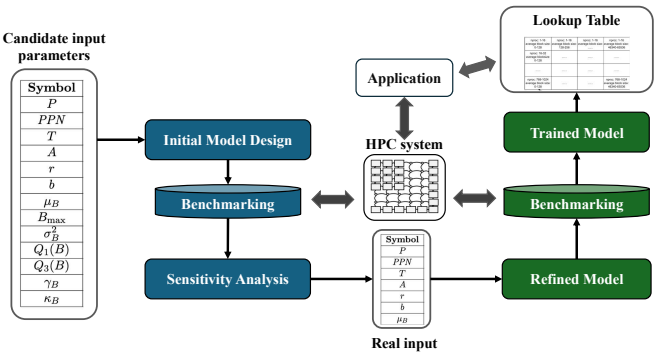

Fig. 1. Overview of the data-driven autotuning framework workflow, comprising phases 1 (blue boxes) and 2 (green boxes).

The second phase (Section V) leverages the refined parameter set and performance data to train multiple ML models, selecting the most effective predictor for our specific optimization problem. The trained model extends the empirical dataset by generating performance predictions for previously unobserved parameter combinations, creating a dense lookup table that covers the entire feasible parameter space. This lookup mechanism integrates seamlessly with existing applications, enabling automatic selection of optimal implementations and parameter configurations with minimal runtime overhead.

IV. PHASE 1: FEATURE SELECTION FOR THE ML MODEL

Feature engineering plays a crucial role in the performance of machine learning (ML) models, as the selection and characterization of input variables directly affect predictive accuracy [22]. In the context of communication optimization, these features must capture the essential properties of parallel workloads and system configurations that shape performance behavior [23]. In our work, we identify a comprehensive set of performance-determining parameters and categorize them into five classes: *execution*, *algorithmic*, *model*, *platform*, and *statistical* parameters, each of which affects a distinct aspect of communication behavior. Table I summarizes these input parameters used in our ML models.

a) Execution Parameters.: Execution parameters are manually configured runtime variables that determine the communication workload and closely relate to the hardware specification. They include the total number of processes (P) and the number of processes per node (PPN). These parameters directly influence communication concurrency, load balance, and message routing.

b) Algorithmic Parameters.: Algorithmic parameters refer to tunable variables specific to an implementation, such as the *radix (r)* and *batch_size (b)* used in our *non-uniform* all-to-all algorithms. They control how the communication rounds are decomposed and how many concurrent communication requests can be issued in each round.

c) Model Parameters.: Model parameters define the high-level algorithmic choices made by the performance model or runtime system. They determine which collective communication algorithm is selected for execution, directly influencing overall performance behavior.

MPI collective primitives are widely adopted but pose performance risks, as their abstraction conceals implementation details. For instance, MPICH [20] provides 3–4 algorithms per primitive, each with distinct performance characteristics depending on various factors. This diversity of algorithmic choices highlights the importance of treating algorithm selection itself as a key model parameter when predicting or optimizing communication performance. In our case, this includes both our proposed *ParAta* and *HieAta* implementations, as well as the vendor-optimized `MPI_Alltoallv`, which we treat as a single baseline despite its internal variants to enable direct comparison.

d) Platform Parameters.: Platform parameters represent architecture-dependent characteristics external to the application, such as the underlying network topology (τ), bandwidth, and latency. Among them, we explicitly include the topology parameter in our model to capture the architectural effects of heterogeneous interconnects.

e) Statistical Parameters.: Statistical parameters describe the non-uniformity of the communication workload. Unlike traditional *uniform* collectives, where message size alone is sufficient, non-uniform all-to-allv patterns require richer descriptors of data distribution. We therefore incorporate multiple statistical descriptors of the data-block size distribution, including the mean (μ_B), maximum ($B_{\max}$), variance (σ_B^2), quartiles ($Q_1(B)$ and $Q_3(B)$), skewness (γ_B), and kurtosis (κ_B). For P processes, B_{ij} denotes the data-block size sent from process i to process j, forming a dense $P \times P$ communication matrix.

The selection of these statistical parameters is motivated by their ability to capture distinct characteristics of data distributions that directly influence the performance of all-to-allv communication. The *mean*, μ_B denotes the average message size for communication between ranks, it characterize the overall amount of data communicated between ranks. The *variance* quantifies the degree of heterogeneity in the message sizes, which can lead to load imbalance. The *maximum* value and *percentile measures* (25% and 75%) characterize both extreme and typical lower and higher values of the message sizes within the distribution. These metrics can potentially enable the model to balance optimization decisions between frequently occurring data sizes and outlier cases that may disproportionately impact overall performance. *Skewness* measures the asymmetry of the distribution, while *kurtosis* characterizes its *tailedness*, indicating the presence and influence of extreme values. These higher-order moments are potentially important for identifying cases in which very large or very small message sizes may introduce performance bottlenecks.

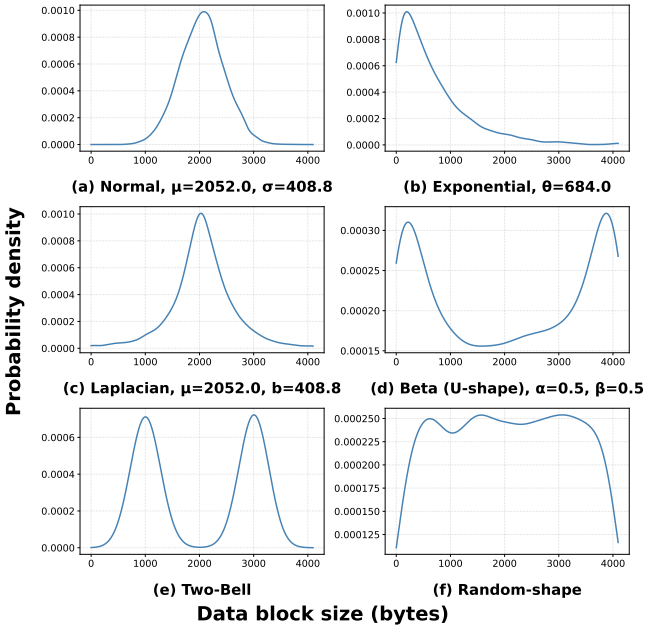

Fig. 2. Example of 6 block-size distributions, each parameterized if applicable, over 4,096 processes, with a maximum block size of 4,096 bytes.

A. Sensitivity Analysis

While we defined comprehensive statistical parameters to characterize various workload distributions, incorporating all parameters into the prediction model is neither feasible nor advisable. The number of input parameters must be constrained for three critical reasons: (1) model complexity and overfitting concerns, (2) lookup table scalability limitations, and (3) runtime overhead considerations.

To address these challenges while retaining the most informative statistical features, we conduct a lightweight sensitivity analysis as a pre-training phase. This sensitivity analysis focuses on identifying the most impactful features rather than constructing a final predictor. To maintain low experimental costs, we employ coarse sampling of non-statistical parameters compared to the comprehensive training process, while extensively varying the underlying data distributions (shown in Figure 2 and explained in Section VI-A) to ensure that retained statistics remain robust across diverse workload patterns. We quantify the input vector:

$$\mathbf{x} = \big(P,\ PPN,\ \tau,\ A,\ r,\ b,\ \mu_B,\ \sigma_B^2,\ B_{\max},\ Q_1(B),\ Q_3(B),\ \gamma_B,\ \kappa_B\big)$$

The predictive model is formally defined as: $\hat{T} = f(\mathbf{x})$, where $\hat{T}$ represents the execution time of a non-uniform all-to-all implementation, and $\mathbf{x}$ denotes a multidimensional feature vector that encapsulates platform, execution, model, algorithmic, and statistical parameters. We quantify each feature's contribution to explained variance using Analysis of Variance (ANOVA). Statistical parameters demonstrating negligible sensitivity are

excluded from the final model to enhance computational efficiency and mitigate overfitting risks.

V. Phase 2: Performance Prediction Framework

Following feature selection, we conduct extensive benchmarking studies (described in Section V-A) and develop machine learning predictors trained on the collected performance data (described in Section V-B), culminating in the development of an optimized offline lookup table for rapid runtime decision-making (outlined in Section V-C). An overview of all components of our framework is illustrated in Figure 1.

A. Benchmarking for Model Fitting

Following the sensitivity analysis (Section IV-A), we focus on fine-grained exploration of system and algorithm parameters while constraining the distribution type only to *random-uniform*. This distribution, where data-block sizes are uniformly sampled between minimum (typically zero) and maximum values, represents the simplest and most widely adopted pattern, yielding an average size equal to half the maximum. This benchmarking strategy strikes an optimal balance between computational cost reduction and comprehensive data collection, which is essential for accurate model training. By restricting the distribution type while conducting significantly more fine-grained sampling of other input parameters compared to the sensitivity analysis phase, we reduce both computational resource requirements while still capturing the essential performance trends required for robust machine learning model development. Runtime measurements are collected for each configuration and subsequently partitioned using a classic $70/10/20$ split into training, validation, and testing datasets to evaluate model generalization performance and prevent overfitting.

B. Machine Learning Model Training

In this work, we evaluated multiple ML techniques for performance prediction. The candidate models included linear regression, Lasso regression, Ridge regression, regression trees, and random forest [24]. For each algorithm, we conducted systematic hyperparameter optimization using cross-validation on the training dataset [24]. Optimal hyperparameters (such as tree depth for regression trees, regularization parameters for Lasso and Ridge, and ensemble size for random forests) were selected based on the performance of the validation set. The final model's performance was assessed using absolute percentage error metrics computed on the held-out test dataset to ensure an unbiased evaluation of its generalization capabilities.

We selected these models due to their effectiveness in handling limited feature dimensionality and computational constraints. They exhibit robust generalization capabilities with reduced susceptibility to overfitting, which is particularly advantageous given the relatively compact feature space inherent to our performance prediction task. The computational efficiency of these algorithms enables rapid convergence during training, making them well-suited for our use case.

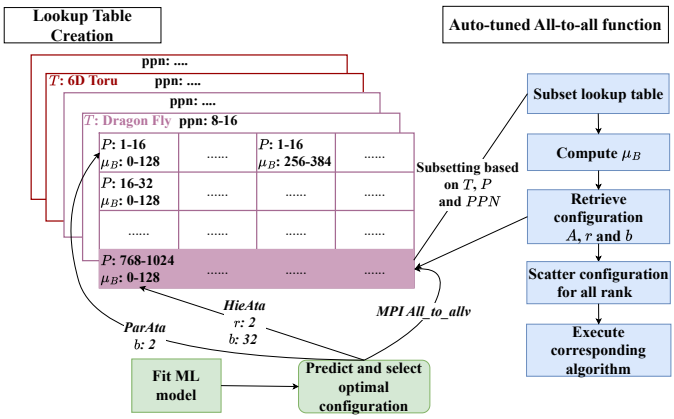

Fig. 3. Runtime flow of lookup table query and algorithm dispatch in the auto-tuned non-uniform all-to-all communication.

C. Lookup Table

To mitigate the computational overhead associated with machine learning model inference during runtime execution, we employ a precomputed lookup table strategy that encapsulates model predictions across a discretized parameter space of input configurations. This lookup table functions as an efficient decision support mechanism, facilitating rapid retrieval of optimal *non-uniform* all-to-all implementations and their corresponding parameters based on observed communication patterns and system-specific characteristics. This lookup table is subsequently integrated with real applications to enable runtime algorithm selection.

Lookup Table Generation: The lookup table is created using a hash table, where the keys correspond to the system parameter (P, ppn, τ) and average block size (μ), and the values correspond to the optimal algorithm implementation along with its optimal configurations (A, r, b). Essentially, the key corresponds to a four-dimensional grid spanning the key input parameters:

$$P \in \{p_1, \ldots, p_m\}, \quad \mathrm{ppn} \in \{\rho_1, \ldots, \rho_k\},$$
$$\mu_B \in \{\mu_1, \ldots, \mu_n\}, \quad \tau \in \{0, 1\},$$

Each cell in the grid, defined as

$$[p_i, p_{i+1}) \times [\rho_\ell, \rho_{\ell+1}) \times [\mu_j, \mu_{j+1}) \times \{\tau\},$$

is associated with a reference point at $(p_i, \rho_\ell, \mu_j, \tau)$. The trained model is then employed to estimate the runtime $(\hat{T})$ for all candidate configurations corresponding to each reference point: $\big(p_i,\ \rho_\ell,\ A,\ r,\ b,\ \mu_j,\ \tau\big)$. For each grid cell, we select the configuration (A, r, b) with the lowest predicted runtime and record it in the corresponding entry of the lookup table. The resulting table contains $(m-1) \times (k-1) \times (n-1) \times 2$ entries, each storing the optimal configuration (A, r, b) for the associated range of $(P, \mathrm{ppn}, \mu_B, \tau)$.

The use of a lookup table not only avoids expensive runtime model evaluations but can also be easily regenerated or adapted for new platforms. This approach bridges the gap between ML-based performance modeling and practical integration within HPC systems, offering a scalable and efficient solution for real-time algorithm selection.

TABLE II
SUMMARY OF INPUT AND OUTPUT PARAMETERS FOR DISTRIBUTIONS

Parameters	Min	Max	Mean	Median	SD
A	–	–	–	–	–
P	32.0	768.0	320.9	256.0	248.1
r	2.0	768.0	66.8	16.0	116.3
$\hat{T}$	0.0	2.1	0.1	0.0	0.2
μ_B	24.5	37752.7	6609.7	2499.9	10186.9
σ_B^2	89.1	213739000.0	14757310.0	2455551.0	45197860.0
$B_{\max}$	51.0	50000.0	11849.1	4954.0	17086.7
$Q_1(B)$	18.0	34581.0	5118.8	1599.8	8491.0
$Q_3(B)$	31.0	41322.0	8101.1	2970.8	12229.5
γ_B	-0.3	0.2	0.0	0.0	0.1
κ_B	-1.3	0.4	-0.4	-0.2	0.5

D. Runtime Integration

At runtime, the application dynamically extracts relevant features from the current execution context and queries the lookup table for a matching entry. Upon successful matching, the associated algorithm and parameters are selected, ensuring fast, informed decision-making with minimal overhead. For unseen configurations, there are two user-configurable resolution strategies. The first employs nearest-neighbor heuristics to identify the most similar configuration within the existing lookup table based on feature proximity metrics. The second strategy implements a fallback mechanism that invokes the original ML model for real-time prediction when necessary.

The detailed steps involved in executing the lookup table to enable autotuning are as follows (also illustrated in Figure 3): First, each rank independently computes the local average data-block size (μ_B). These local μ_B values, together with the known P, ppn, and topology flag τ, are then transmitted to a designated master process, which subsequently calculates the global μ_B. Using the parameter tuple $(P, \text{ppn}, \mu_B, \tau)$, the master process identifies the corresponding entry in the precomputed lookup table and retrieves the associated algorithm and tuning parameters (A, r, b). These parameters are then broadcast from the master process to all processes. Finally, the selected all-to-allv algorithm A is executed across all processes using the retrieved parameters r and b.

Overall, the process of using the lookup table in an application incurs minimal overhead, as it involves only a lightweight reduce and scatter operation, a small number of integer comparisons, and a single table lookup.

Integration into MPI libraries. At present, our autotuning framework is implemented as a C++ wrapper around `MPI_Alltoallv`, allowing it to intercept calls, analyze data distributions, and select the best algorithm and parameters with minimal overhead. While this standalone design simplifies prototyping, our long-term goal is to integrate it into production MPI libraries such as MPICH and Open MPI. Integration would involve adding our implementations to the `coll/alltoallv` module, registering them in `coll_algorithms.txt`, exposing tunable parameters as MPI_T control variables (CVARs), and extending the collective selection (CSEL) engine to use our ML-based lookup table. This would enable applications to benefit from optimized non-uniform Alltoallv automatically, without code changes.

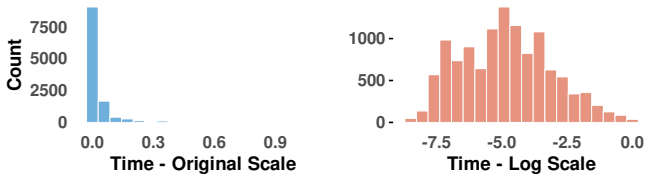

Fig. 4. Histograms of non-uniform all-to-all runtimes on the original scale (left) and the log-transformed scale (right).

VI. EVALUATION

We evaluate our framework on two systems: **Polaris** at Argonne Leadership Computing Facility (ALCF) and **Fugaku** at RIKEN R-CCS. Polaris consists of 560 compute nodes, each equipped with dual-socket 32-core AMD CPUs and 4 NVIDIA A100 GPUs, connected by a Slingshot Dragonfly network. The software environment is configured with Cray MPICH version 8.1.16. Fugaku comprises 158,976 nodes, each with 48 user-accessible A64FX cores and a 6D-torus Tofu-D interconnect. Its software environment is based on Fujitsu MPI 4.12.0, derived from OpenMPI.

We conducted two benchmarking phases to collect training data for our machine learning models. The first phase gather data for sensitivity to identify key parameters, and the second phase collected data for model training. Together, the experiments used about 1,370 node hours. In each experiment, we changed one input parameter at a time (see Section IV), such as varying the process count P or the number of processes per node (*PPN*). Each configuration was run five times, and we used the median runtime to reduce noise. The evaluated algorithms include *ParAta*, *HieAta* (see Section II-C), and the vendor-optimized `MPI_Alltoallv`.

A. Performance of Sensitivity Analysis

The sensitivity analysis was designed to identify the most impactful statistical features affecting prediction accuracy. To optimize time and computational resource utilization, we employed coarse sampling of non-statistical parameters (detailed in Section IV-A).

Benchmarking: In this phase, we systematically varied the process count P from 32 to 4,096, doubling the values between successive experimental configurations. The number of processes per node (*PPN*) was varied from 4 to 32. For the radix parameter, we evaluated three specific values: 2, $\sqrt{P}$, and P, which have been proven to demonstrate optimal efficiency for short (<512 bytes), medium (<4,096 bytes), and long messages, respectively [2]. Additionally, the *batch_size* was fixed to the number of cores per node (32) on Polaris, as this parameter has minimal impact on the sensitivity analysis.

For each experimental run, we generated per-rank block sizes according to multiple distributions, including random-uniform, normal, Poisson, exponential, and Laplacian distributions. To improve the generalizability of our analysis, we also incorporated randomly generated undefined distributions. All distributions were constrained to generate random values within the range of 1 and the maximum size of data-blocks

TABLE III
ANOVA ANALYSIS FOR STATISTICAL PARAMETERS (DF: DEGREES OF FREEDOM).

Parameters	Df	Sum of Squares	Mean Squares	F value	p-value
A	3	5404	2702	1.371e+04	<2e-16
P	1	51666	51666	2.622e+05	<2e-16
r	1	1284	1284	6517	<2e-16
μ_B	1	265153	265153	1.346e+06	<2e-16
σ_B^2	1	334	334	1697	<2e-16
$B_{\max}$	1	99	99	504.4	<2e-16
$Q_1(B)$	1	150	150	763.0	<2e-16
$Q_3(B)$	1	22	22	110.8	<2e-16
γ_B	1	9	9	45.60	1.46e-11
κ_B	1	10	10	52.86	3.63e-13

(M). M was varied from 8 bytes to $8,192$ bytes. Additionally, since each distribution has its own characteristics, we employed three to four distinct configurations per distribution. Representative examples of these distributions are illustrated in Figure 2. For each distribution, we computed the corresponding statistical parameters as defined in Table I and measured the runtime performance across different configurations to conduct the sensitivity analysis.

Data processing: The initial dataset collected from benchmarking exhibited significant skewness, necessitating data preprocessing before conducting sensitivity analysis. First, we computed values for predefined input and output (runtime $\hat{T}$) parameters (see Table I). These metrics were then compiled into an aggregated summary, as presented in Table II.

Subsequently, we generated histograms of these parameters to visually examine their distributions. These plots clearly demonstrated substantial skewness, as illustrated in Figure 4, which poses challenges for both sensitivity analysis and accurate model training. Ideally, each parameter should exhibit a uniform or normal (bell-shaped) distribution to facilitate reliable sensitivity analysis conclusions and enable the development of accurate machine learning models for performance prediction. To address this issue, we employed a logarithmic transformation for strictly positive parameters to stabilize and normalize their distributions. As demonstrated in Figure 4, the transformed parameters were re-plotted to confirm that their distributions became more balanced and suitable for subsequent analysis.

Experimental results: Figure 5 presents the correlation heatmap between candidate input parameters and the output parameter (runtime $\hat{T}$). The analysis reveals that several statistical parameters, including μ_B, $B_{\max}$, $Q_1(B)$ and $Q_3(B)$, exhibit strong and approximately equivalent correlations with $\hat{T}$ (runtime). In contrast, higher-order moments, σ_B^2, γ_B, and κ_B, demonstrate weaker correlations with execution time. To avoid unnecessary computational overhead, we retained only the four most predictive metrics for further analysis. However, incorporating four statistics would still significantly increase the dimensionality of the lookup table. We then conducted an analysis of variance (ANOVA) [25], a statistical method used to determine whether one or more input parameters have a statistically significant effect on the output parameter ($\hat{T}$).

Table III presents the ANOVA results for each statistical

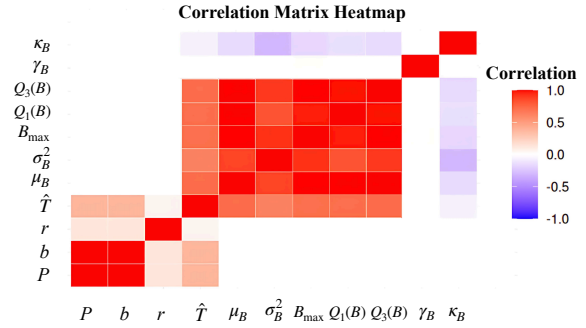

Fig. 5. Correlation matrix heatmap of candidate statistical parameters versus measured runtime ($\hat{T}$). Darker red indicates a higher correlation.

parameter, where several components are used to evaluate the importance of a parameter. We used the *Sum of Squares* metric to quantify the proportion of variance in the output parameter explained by each statistic. The table shows that the μ_B parameter has the highest *Sum of Squares* (265,153), significantly exceeding the others. This result suggests that the μ_B is the most influential statistic and a strong candidate for retention in the predictive model. Therefore, we use only μ_B in our ML model and lookup table. This simplifies the model, reduces overfitting, speeds up table generation, and lowers runtime overhead.

B. Performance of ML Training

Following the sensitivity analysis, we selected the average size of data-blocks (μ_B) as the sole statistical parameter. Consequently, the input vector $\mathbf{x}$ is refined as: $\mathbf{x} = (P,\ PPN,\ \tau,\ A,\ r,\ b,\ \mu_B)$

Benchmarking: For machine learning model training, we employed significantly finer sampling compared to the sensitivity analysis to acquire sufficient training data. Additionally, rather than utilizing multiple distributions, we adopted the most widely used distribution, random-uniform, which ensures that sample data are uniformly distributed between 0 and the *maximum data-block size*. In this benchmarking phase, we varied the process count P from 32 to 4,096. For the *maximum data-block size*, we extended the range from 16 bytes to 524,288 bytes. The *batch_size* (b) was varied from 1 to 32, and the *radix* (r) was varied from 2 to P. The values of all these parameters were doubled at each increment. Additionally, the value of radix was sampled more finely between three critical points: 2, $\sqrt{P}$, and P. Following data collection, we partitioned the dataset into 70% for training, 10% for validation, and 20% for testing. This training data collected is also processed using the same transformation strategy presented in the previous sensitivity analysis section.

For every process count and message size configuration we plot the best performing algorithm and its corresponding parameter. As shown in Figure 6, each tile represents a (P, max-msg) configuration, labeled by the fastest algorithm and parameters (**M** = vendor `MPI_Alltoallv`, **P–*r*** = ParAta, **H–*r*–*b*** = HieAta). The figure shows that the optimal configuration varies across process counts and message sizes, and the patterns differ between Polaris and Fugaku due to

TABLE IV
COMPARISON OF ABSOLUTE % ERROR ON TEST SET FOR ML MODELS.

Linear	Lasso	Ridge	Decision Tree	Random Forest	MLP
48.52	48.38	48.48	**2.22**	**1.91**	6.11

TABLE V
ESTIMATED COST BREAKDOWN OF FRAMEWORK CONSTRUCTION AND RUNTIME OVERHEAD. TRAINING COSTS ARE ONE-TIME, WHILE RUNTIME LOOKUP REPEATS PER INVOCATION.

Component	**Cost / Overhead**
Benchmarking for sensitivity analysis (Polaris only)	∼163 node hours
Benchmarking for ML training (each platform)	∼445.8 node hours
Model training	∼0.5 hours (single node)
Lookup table generation	∼0.75 hours (single node)
Runtime lookup (C++ wrapper)	$< 1\%$ overhead per call

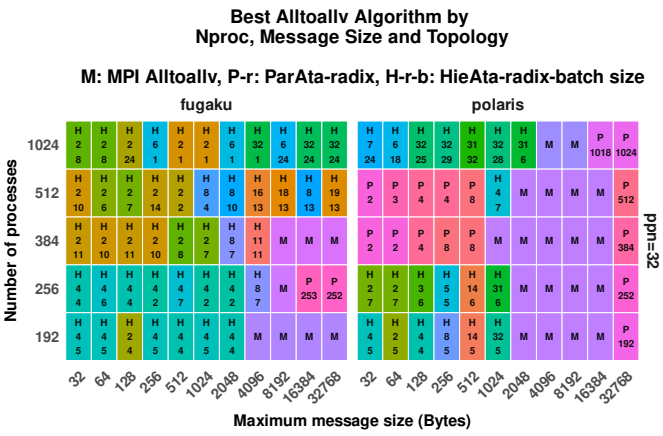

Fig. 6. Optimal algorithm configuration for non-uniform `MPI_Alltoallv`. Each tile references a system configuration of a different number of processes, maximum message size, and network topology. The optimal algorithm configuration varies with P and message size, and the patterns differ between Polaris (dragonfly) and Fugaku (6D torus).

their different network topologies. These benchmarking results further indicate the complex relationship between performance and the input parameter space, further amplifying the need of a data-driven model, to model this complex space.

Experimental results: Our training models included linear regression, Lasso regression, Ridge regression, regression trees, and random forest. After training these models, we evaluated their performance using absolute percentage errors on the test set, as presented in Table IV. The results demonstrate that linear, Lasso, and Ridge regressions produced high prediction errors (exceeding 48%), whereas regression tree and random forest models achieved significantly lower errors (below 2.5%). Notably, the regression tree model exhibited the low prediction error (2.22%) while maintaining the highest computational efficiency among all evaluated models.

Framework Cost Breakdown: Constructing our autotuning framework required a one-time offline cost. Benchmarking for sensitivity analysis was performed on Polaris, while large-scale benchmarking for ML training was conducted on both Polaris and Fugaku, followed by model fitting and lookup table generation. As summarized in Table V, most of the time was spent on benchmarking, with model training and table generation occupying only a small fraction. At runtime, the MPI C++ wrapper adds negligible overhead, limited to computing global statistics and a single table lookup.

VII. APPLICATION

In this section, we evaluate the performance of our lookup table using a real-world application: transitive closure computation, a foundational graph mining problem [26].

Application-graph mining: Transitive closure (TC) is a widely used graph mining problem that can be computed using a classic fixed-point approach, which repeatedly applies a relational algebra (RA) kernel to a graph G. This operation incrementally discovers paths of increasing length within the graph and continues until no new paths are found, thereby reaching a fixed point. We employ an MPI-based open-source library for parallel relational algebra [26], which utilizes `MPI_Alltoallv` in each iteration of the fixed-point loop to perform data shuffling. The alternative non-uniform all-to-all algorithms included in this work preserve the same function signature as `MPI_Alltoallv`, allowing them to be seamlessly substituted. For our experiments, we use a graph with $1{,}014{,}951$ edges, obtained from the SuiteSparse Matrix Collection [27]. This graph undergoes over $5{,}800$ iterations of all-to-allv communication before reaching the fixed point.

Lookup table validation: We evaluate the performance of our autotuning framework using both Polaris (Dragonfly) and Fugaku (6D torus) supercomputers, evaluating process counts of 200, 256, 400, 512, 600, and 1024, including non-power-of-two cases. The experimental results in Table VI show that our autotuning framework consistently outperforms the standard `MPI_Alltoallv` implementation across all configurations. On Polaris, the communication time is reduced by $2.4\times$–$6.0\times$, leading to 1%–12.8% shorter total runtimes. This proves the effectiveness of our auto-tuning framework on speeding up the MPI application.

In addition to the direct performance comparison, we also examined how our autotuner selects algorithms and parameters over the execution of the application (512 processes on Polaris). Figure 7 plots the chosen algorithm against the average block size at each iteration. When the block size is relatively large, the tuner prefers the standard `MPI_Alltoallv`. In contrast, during iterations with smaller blocks, particularly at the start and in the latter half, it switches to our *HieAta* and *ParAta* implementations (as shown in rows 1–2 of 7). Moreover, the autotuner adapts each algorithm's parameters on the fly. For *HieAta*, a radix of 2 is selected under small block sizes, effectively mimicking a Bruck-style algorithm that excels in small block sizes. As block sizes grow (and *HieAta* remains the best choice), the radix increases accordingly, and the *batch_size* is also adaptively selected to cooperate with the selected radix (as shown in rows 3–4 of 7). This visualization shows that our autotuner, powered by a compact, data-driven lookup table, can automatically pick both the

TABLE VI
PERFORMANCE COMPARISON OF OUR AUTOTUNING FRAMEWORK AGAINST THE STANDARD MPI_ALLTOALLV IMPLEMENTATION ON THE TRANSITIVE-CLOSURE APPLICATION. SPEEDUP = OFFICIAL/ OURS.

Polaris (Dragonfly network)			
P	Implementation	Comm. Time (s)	Total Time (s)
200	Official MPI_Alltoallv	8.69	353.92
200	Auto-tuned (*Ours*)	3.57	349.24
200	Speedup (Off/Ours)	2.43×	1.01×
256	Official MPI_Alltoallv	11.68	297.64
256	Auto-tuned (*Ours*)	3.93	289.94
256	Speedup (Off/Ours)	2.97×	1.03×
400	Official MPI_Alltoallv	13.14	195.25
400	Auto-tuned (*Ours*)	3.39	185.15
400	Speedup (Off/Ours)	3.88×	1.05×
512	Official MPI_Alltoallv	14.02	149.47
512	Auto-tuned (*Ours*)	3.60	139.28
512	Speedup (Off/Ours)	3.89×	1.07×
600	Official MPI_Alltoallv	11.20	131.74
600	Auto-tuned (*Ours*)	1.89	123.25
600	Speedup (Off/Ours)	5.93×	1.07×
1024	Official MPI_Alltoallv	11.52	79.50
1024	Auto-tuned (*Ours*)	1.91	69.49
1024	Speedup (Off/Ours)	6.03×	1.14×

optimal algorithm and its parameters at runtime, minimizing all-to-allv communication time without intervention.

VIII. RELATED WORK

Due to the complexity of hardware, communication topologies, and workloads, static tuning methods for collective operations often perform suboptimally. Recent research increasingly incorporates ML techniques to automate and optimize collective algorithms for diverse communication scenarios.

Pellegrini et al. [10] present an early ML approach for this domain, training regression models on benchmark performance metrics to predict optimal parameters from execution characteristics (message sizes, process counts). While demonstrating significant improvements over default configurations, their offline training methodology limits scalability to complex applications and dynamic runtime environments. Hunold and Carpen-Amarie [11] introduced a decision tree-based framework for collective algorithm selection, demonstrating through benchmarking analysis that ML-based selection outperforms traditional heuristics. Zheng et al. [17] similarly applied ML to model and optimize MPI collectives, achieving substantial performance gains on modern HPC systems. Wilkins et al. developed FACT [14], which enhances generalization across diverse communication scenarios, and ACCLAiM [13], which integrates auto-tuning into production workflows via hybrid offline/online ML strategies. Han et al. [16] introduced PML-MPI, a pre-trained ML framework for runtime collective selection achieving robust generalization with minimal tuning overhead. OMPICollTune [15] addresses the limitations inherent to static and offline modeling approaches by using incremental online learning. Their system continuously refines its predictive model during program execution, enabling adaptation to application phase transitions and evolving communication

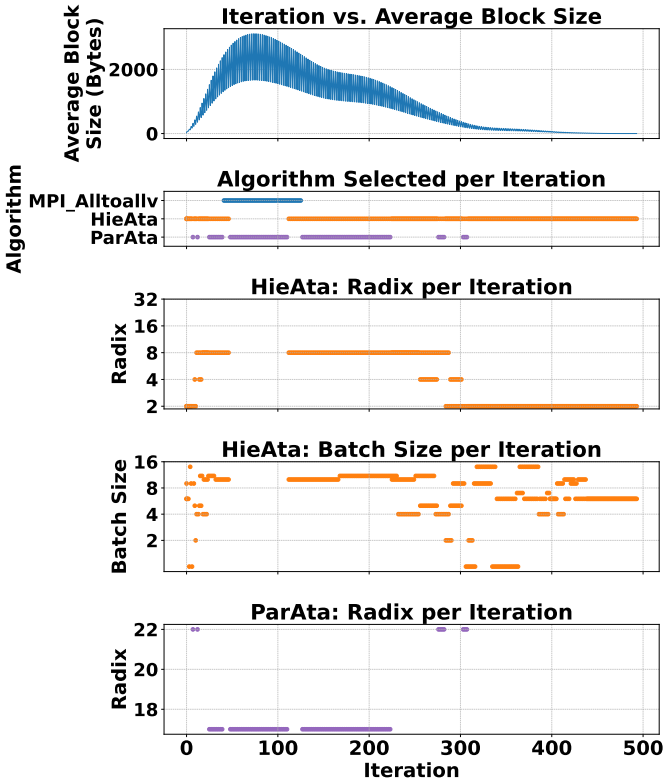

Fig. 7. Visualization of dynamic selection of algorithm and parameters across iterations. The x-axis corresponds to temporal iterations.

patterns. Hunold et al. [12] predicts MPI collective performance using regression models (XGBoost, GAM) trained on benchmark data with algorithmic and runtime parameters, enabling runtime retrieval of optimal configurations directly from the fitted model. Unlike [12] and OMPICollTune [15], our approach focuses on all-to-allv and leverages statistical features of message-size distributions, combining offline ML modeling with lightweight runtime lookup for adaptive, low-overhead tuning.

Although not explicitly ML-based, Nuriyev and Lastovetsky [7] offer a valuable comparison by using analytical performance modeling for dynamic selection of optimal MPI algorithms at runtime. While less flexible than ML in modeling non-linear interactions, analytical methods provide transparent, interpretable decision frameworks with potentially lower computational overhead.

IX. CONCLUSION AND FUTURE WORK

In this paper, we presented a data-driven autotuning framework for non-uniform MPI_Alltoallv that automatically selects among different implementations at runtime. By combining sensitivity analysis with machine learning, we built a compact lookup table that maps simple statistics of the block-size distribution and process count to the best algorithm configuration. Our C++ auto-tuned function achieves a speedup of up to 6.03× in communication time and reduces total runtime by 12.6% on a 1024-rank graph mining problem.

One key limitation of our approach is the extensive offline benchmarking required to train the performance models. While this one-time cost does not affect runtime overhead, it may limit broader applicability. Another limitation of our work is that changes in compiler toolchains and MPI runtimes between the benchmarking and application environments could lead to sub-optimal selection of All-to-allv algorithm configuration for the MPI application. Future work will explore data-efficient techniques, such as active learning, transfer learning, and online tuning, to enable adaptation to new workloads and architectures with significantly less prior measurement.

While our study focused on non-uniform `MPI_Alltoallv`, the framework can be extended to other collectives and applications as well. Future work will assess whether this sensitivity analysis and machine learning approach can be generalized to other collective operations and irregular communication patterns. We also plan to integrate analytical and roofline models to evaluate how closely our auto-tuned configurations approach theoretical performance limits.

X. Acknowledgement

This work was funded in part by NSF PPoSS grant CCF-2316157 and NSF SHF grant CCF-2401274. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the Polaris supercomputer located at the Argonne National Laboratory. We also extend our thanks to RIKEN for granting access to computing time on Supercomputer Fugaku hosted at the RIKEN Center for Computational Science. Venkatram Vishwanath and Michael E. Papka are supported by the Office of Science, U.S. Department of Energy, under contract DE-AC02-06CH11357.

References

[1] K. Fan, S. Petruzza, T. Gilray, and S. Kumar, "Configurable algorithms for all-to-all collectives," in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, 2024, pp. 1–12.

[2] K. Fan, J. Domke, S. Ba, and S. Kumar, "Parameterized algorithms for non-uniform all-to-all," in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '25. New York, NY, USA: Association for Computing Machinery, 2025.

[3] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 93–104.

[4] M. T. Hussain, O. Selvitopi, A. Buluç, and A. Azad, "Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 90–100.

[5] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout, "Pidx: Efficient parallel i/o for multi-resolution multi-dimensional scientific datasets," in *2011 IEEE International Conference on Cluster Computing*, 2011.

[6] K. Fan, T. Gilray, V. Pascucci, X. Huang, K. Micinski, and S. Kumar, "Optimizing the bruck algorithm for non-uniform all-to-all communication," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 172–184.

[7] E. Nuriyev and A. Lastovetsky, "Accurate runtime selection of optimal mpi collective algorithms using analytical performance modelling," 2020. [Online]. Available: https://arxiv.org/abs/2004.11062

[8] 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.

[9] T. Worsch, R. Reussner, and W. Augustin, "On benchmarking collective mpi operations," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2002, pp. 271–279.

[10] S. Pellegrini, T. Fahringer, H. Jordan, and H. Moritsch, "Automatic tuning of mpi runtime parameter settings by using machine learning," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, ser. CF '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 115–116.

[11] S. Hunold and A. Carpen-Amarie, "Algorithm selection of mpi collectives using machine learning techniques," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 45–50.

[12] 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)*, 2020, pp. 259–269.

[13] 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)*, 2022, pp. 161–171.

[14] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2021, pp. 36–45. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ExaMPI54564.2021.00010

[15] 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.

[16] M. Han, G. K. Reddy Kuncham, B. Michalowicz, R. Vaidya, M. Abduljabbar, A. Shafi, H. Subramoni, and D. K. D. Panda, "Pml-mpi: A pre-trained ml framework for efficient collective algorithm selection in mpi," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024, pp. 761–770.

[17] W. Zheng, J. Fang, C. Juan, F. Wu, X. Pan, H. Wang, X. Sun, Y. Yuan, M. Xie, C. Huang, T. Tang, and Z. Wang, "Auto-tuning mpi collective operations on large-scale parallel systems," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 670–677.

[18] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on parallel and distributed systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[19] Q. Kang, R. Ross, R. Latham, S. Lee, A. Agrawal, A. Choudhary, and W.-k. Liao, "Improving all-to-many personalized communication in two-phase i/o," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.

[20] https://www.mpich.org, MPICH Home Page.

[21] https://www.open-mpi.org, OpenMPI Home Page.

[22] E. Sullivan, "Understanding from machine learning models," *The British Journal for the Philosophy of Science*, 2022.

[23] J. Brownlee, "How to choose a feature selection method for machine learning," *Machine Learning Mastery*, vol. 10, pp. 1–7, 2019.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] D. C. Montgomery, *Design and Analysis of Experiments*, 9th ed. Hoboken, NJ, USA: John Wiley & Sons, 2017.

[26] S. Kumar and T. Gilray, "Distributed relational algebra at scale," in *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019.

[27] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.