Practical Machine Learning Autotuning for Large-Scale Collective Communication

Michael Wilkins, Yanfei Guo, *Member, IEEE*, Rajeev Thakur, *Fellow, IEEE*, Peter Dinda, *Fellow, IEEE*, and Nikos Hardavellas, *Member, IEEE*

***Abstract*—Collective communication is a fundamental communication model for parallel computing on distributed memory systems. The performance of a collective operation depends on the underlying algorithm. Machine learning (ML)–based autotuners can optimize algorithm selection to enhance collective performance. Previous approaches are impractical for production use at large scale, however, due to prohibitively long training times that exceed typical job durations. This paper introduces ACCLAiM (Advancing Collective Communication Autotuning using Machine Learning), the first ML-based collective algorithm selection autotuner capable of accelerating production applications on large-scale supercomputers. ACCLAiM incorporates several improvements over prior designs in training point selection, handling of non-power-of-two feature values, model validation, and data collection. Our approach leverages variance-based active learning alongside topology-aware benchmark parallelization to eliminate unnecessary training points and maximize machine utilization, thereby significantly reducing training time. We present ACCLAiM as an open-source prototype and provide a comprehensive experimental evaluation. We demonstrate how each of ACCLAiM's enhancements contributes to a substantial reduction in training time compared with the previous state-of-the-art approach, cumulatively reducing the training time to 5–10 minutes even at large scale. We demonstrate ACCLAiM's usefulness on two leadership-class supercomputers and showcase its practical benefits for applications, achieving speedups up to 4.1x.**

***Index Terms*—Autotuning, collective communication, MPI, machine learning.**

I. Introduction

THE Message Passing Interface (MPI) is the de facto standard communication protocol for high-performance computing (HPC) applications. Within MPI, the most popular primitives are collective operations (i.e., collectives). Collectives coordinate data exchange among all processes in a group, such as *MPI_Allreduce* and *MPI_Bcast*. Collectives are crucial yet often become bottlenecks, consuming up to 80–90% of execution time in artificial intelligence (AI) and scientific workloads [1], [2].

The performance of a collective operation is highly dependent on the choice of the underlying algorithm. A suboptimal choice can degrade performance by 35–40% [3]. Selecting the optimal algorithm is challenging, however, because of factors such as message size, network latency, and unpredictable network congestion. This complexity has spurred interest in automated tools for algorithm selection.

Previous studies have explored automatic tuning of collective algorithms using analytical models and exhaustive benchmarking [4], [5], [6], [7], [8], [9]. Analytical models are difficult to implement and maintain, and exhaustive benchmarking is impractical for large systems because of its need for frequent reruns. Thus, a scalable optimization tool for supercomputers is still needed.

Machine learning (ML) presents a promising alternative [10]. ML autotuners use microbenchmarks to learn algorithm performance characteristics and predict outcomes for unseen scenarios, capturing real-time variations such as network congestion. This approach reduces the benchmarking burden compared with exhaustive methods.

Despite promising results in simulations [3], [10], [11], current ML autotuners are hindered by extensive training times. For instance, training a model on the Theta supercomputer could take up to 24 hours, matching the maximum job length and thus negating any performance gains [11]. Simulations also overlook real-world factors such as non-power-of-two inputs, further complicating training.

This paper addresses these challenges by identifying key bottlenecks in existing ML autotuners: training data selection, non-power-of-two inputs, model testing, and data collection. We propose solutions such as jackknife variance calculations [12] and topology-aware parallel data collection to optimize training efficiency. Our approach minimizes training points and maximizes hardware utilization.

We introduce ACCLAiM (Advancing Collective Communication Autotuning using Machine Learning), the first ML autotuner practical for large-scale production systems. We evaluate ACCLAiM's ability to tune collective algorithm selection at scale on Aurora [13], an exascale supercomputer, and verify its generality through testing on Polaris [14]. Our results show that ACCLAiM has a training time per collective of 5–10 minutes using up to 2048 nodes, providing net speedups for jobs as short as

Received 11 August 2025; revised 4 January 2026; accepted 21 January 2026. Date of publication 6 February 2026; date of current version 16 March 2026. This work was supported in part by the Laboratory Directed Research and Development (LDRD) from Argonne National Laboratory, provided by the Director, Office of Science, through the U.S. Department of Energy under Contract DE-AC02-06CH11357, and in part by the National Science Foundation under Award CCF-2119069, Award CCF-2119352, Award CCF-2028851, Award CCF-2028921 and Award CCF-2028958. Recommended for acceptance by J. Carretero. *(Corresponding author: Michael Wilkins.)*

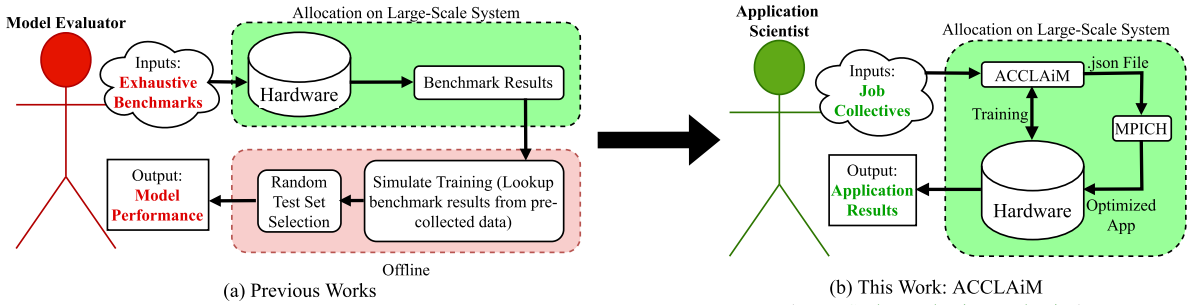
Michael Wilkins is with Cornelis Networks, Inc, Wayne, PA 19087 USA (e-mail: mjwilkins18@gmail.com).

Yanfei Guo is with NVIDIA Corporation, Santa Clara, CA 95051 USA.

Rajeev Thakur is with Argonne National Laboratory, Lemont, IL 60439 USA.

Peter Dinda and Nikos Hardavellas are with Northwestern University, Evanston, IL 60208 USA.

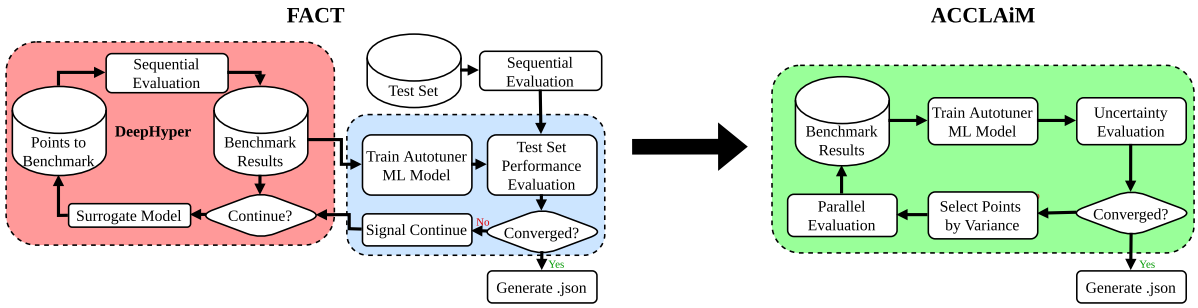
Digital Object Identifier 10.1109/TPDS.2026.3661876



(a) Previous Works (Simulated Evaluation)

(b) This Work: ACCLAiM (Large-Scale, Production Evaluation)

Fig. 1. Previous works vs. ACCLAiM. Previous works use offline simulation to show the promise of ML collective autotuners. ACCLAiM uses an improved approach that is practical for application developers on production supercomputers.



(a) Previous State-of-the-Art Training Methodology

(b) Proposed Training Methodology

Fig. 2. Previous FACT methodology vs. new ACCLAiM methodology. FACT's training methodology (blue) contains multiple inefficiencies and relies on another ML-based tool, DeepHyper (red). ACCLAiM (green) improves the training process, greatly reducing the training time.

20–30 minutes when collectives dominate runtime. To prove its usefulness, we present two production applications (GAMESS and SHAMROCK) that benefit greatly from ACCLAiM's tuning, up to a $4.1\times$ processing rate improvement.

Figs. 1 and 2 showcase ACCLAiM's novelty compared with previous work. We highlight how ACCLAiM advances from a simulated environment to production usefulness via its enhanced training methodology.

Our contributions are as follows:

- A training point selection methodology that uses variance calculations and guided sampling to minimize data requirements and handle non-power-of-two features
- A model testing procedure that leverages cumulative variance to eliminate additional data collection
- Topology-aware parallel data collection that greatly improves machine utilization during training
- ACCLAiM, an ML collective autotuner that can accelerate HPC applications on large-scale supercomputers
- A thorough empirical evaluation conducted on two leadership-class supercomputers, Aurora and Polaris
- Experiments with real-world, production HPC applications that significantly benefit from ACCLAiM tuning (e.g., up to $4.1\times$ speedup for SHAMROCK)

II. BACKGROUND

We introduce our evaluation environments and the significance and challenges of selecting collective algorithms, as well as the design and limitations of current ML approaches. Non-ML solutions are discussed in Section IX.

A. Evaluation Environments

This work utilizes both simulated and real-world experiments on production systems. For direct comparison with existing work in our intermediate results, we use the testing framework in Fig. 1(a) from our previous work [11]. To perform a "benchmark run," we look up the corresponding value in a pre-collected dataset, which includes exhaustive benchmarking results (i.e., the performance of every algorithm for every scale and message size). We use the same pre-collected dataset as the previous work to create a fair comparison. This data was collected by using up to 64 nodes, each containing an Intel Xeon-E5-2694v4 with 36 cores (of which the dataset uses up to 32) and 128 GB of DDR4 memory. The maximum message size in the dataset is 1 MB.

For our comprehensive evaluation of ACCLAiM, we perform experiments primarily on Aurora, a leadership-class supercomputer at Argonne National Laboratory. The full machine consists of 10,624 nodes, each with two Intel CPU Max Series and six Intel Data Center GPU Max Series. In total, each node has 104 physical CPU cores, 12 logical GPU tiles, 1,024 GB of DDR5 memory, 128 GB of high-bandwidth memory, and eight Slingshot 11 network cards connected via a dragonfly topology.

To ensure the generality of ACCLAiM, we also perform experiments on Polaris, another large-scale HPC system at Argonne. Polaris nodes are equipped with a single AMD CPU

with 4 NVIDIA A100 GPUs, again connected with a Slingshot 11 dragonfly network.

B. Collective Algorithm Selection

By default, the open-source implementations of the MPI standard—Open MPI [15], MVAPICH [16], and MPICH [17]—use heuristics to make selections. This work focuses on MPICH because it serves as the basis of many popular production MPI libraries and exposes its algorithm selection for tuning via. json files. Because collective operations alone make up a significant portion of HPC application runtimes [1], [2], [18], improving collective performance directly improves application performance.

Selecting the optimal collective algorithm is challenging because of the numerous influential variables involved, which can be divided into programmatic and non-programmatic categories [11]. Programmatic variables are inputs directly related to the collective call, such as message size, number of processes, and number of processes per node. Non-programmatic variables are factors not visible to the programmer, including CPU architecture, network topology, latency, and congestion.

To manually select the best algorithm for a given scenario, developers must understand the impact of both programmatic and non-programmatic variables. For instance, consider choosing between MPICH's two algorithms for MPI_Reduce. The first algorithm, $binomial$, constructs a binomial tree where each node reduces its children's values and forwards the result to the "parent." The second algorithm, $scatter\_gather$, operates in two phases: first, a "scatter" distributes the complete reduction across n processes, each handling $1/n$ of the values; then, the root node "gathers" the fully reduced values.

Conventional wisdom suggests that $binomial$ is preferable for small message sizes because of fewer sequential steps, while $scatter\_gather$ is better for large message sizes because it maximizes bandwidth utilization. This logic fails, however, when considering non-programmatic variables such as network factors. For example, job launchers such as PBS and Slurm typically do not guarantee the proximity of nodes allocated to a job, leading to higher communication latency. In high-latency networks, $binomial$ is advantageous even for large message sizes because of fewer, larger communications that are less affected by latency. Conversely, $scatter\_gather$ excels in low-latency environments with its numerous smaller communications.

This is just one example; other factors, such as network congestion, further complicate the effective network latency. The design space for collective algorithm selection is too complex to navigate manually. ML collective autotuners learn patterns in algorithm performance, automatically optimizing collective operations.

C. Dynamism of Non-Programmatic Values

Simulated experiments have demonstrated that ML autotuners can grasp performance trends influenced by numerous non-programmatic variables. Since these simulations rely on pre-collected benchmark data, however, it remains uncertain whether the models retain their accuracy amidst highly dynamic variables. For instance, varying effective topologies can significantly impact algorithm performance for each job, meaning that an ML autotuner trained during one job might not predict accurately for another. To address the challenge posed by dynamic variables, ML collective autotuners must be retrained at the start of every job. This requirement sets a stringent standard for training time, as each job using the autotuner must recover the training cost within a single application run. On the positive side, this necessity decentralizes the design, enhancing usability. Instead of coordinating a centralized autotuner across multiple users or applications, application scientists can independently train their own models. This approach allows any user to leverage ML collective autotuning without needing special permissions.

D. Existing ML Autotuner Approaches

In this study, we adopt the state-of-the-art ML collective autotuners as our baseline design. [3] pioneered the first ML autotuner, which utilized a distinct ML model for each algorithm within a collective operation. For instance, if an MPI implementation included 4 collective operations, each with 3 algorithms, the autotuner would develop 12 separate ML models. Each model is tasked with predicting the performance of a specific algorithm, accepting inputs including the number of processes, number of processes per node, and message size—referred to as "feature values." The complete set of possible input values is known as the "feature space," and each model outputs a predicted execution time in microseconds.

We adopt this approach with modifications, opting to train a single ML model per collective operation instead of multiple models for each algorithm (e.g., 4 separate ML models in the example above). This streamlined design enhances scalability, especially when dealing with a larger number of algorithms.

The autotuner gathers training data through microbenchmarks, which in [3] are randomly sampled from the feature space on small clusters (number of nodes ≤ 48). To select an algorithm, the autotuner queries the models for each algorithm and chooses the one with the lowest predicted execution time. [3] demonstrated that the autotuner's selections outperformed the defaults by 35–40%.

Recognizing the scalability limitations of the original training method for larger machines due to the high cost of benchmarking random points, we introduced the FACT methodology, which employs active learning to intelligently select training points [11]. Active learning iteratively selects new training points to efficiently train the ML model. In simulated experiments, FACT achieved near-optimal performance with 6.88x less training data collection time compared to [3].

Despite its enhanced performance, FACT presents several remaining bottlenecks that limit its practicality. It still requires up to 24 hours of training time for larger-scale systems (128–512 nodes), which remains impractical [11]. Most of this time is spent on training data collection. Fig. 2 illustrates FACT and contrasts it with our proposed ACCLAiM methodology.

E. Comparison Technique

Throughout this work, we employ our previous performance comparison framework to evaluate autotuners [11]. This technique utilizes the *average slowdown* metric, which measures

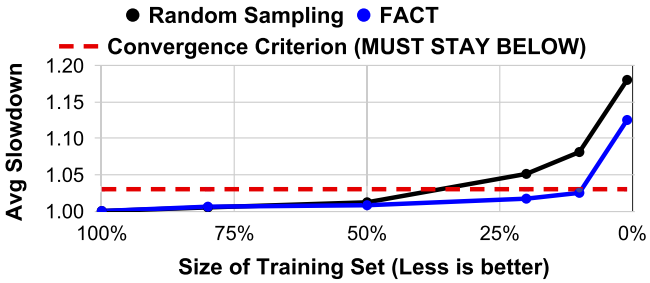


Fig. 3. Comparison of two prior ML autotuner approaches: random sampling [3] and active learning (FACT) [11]. FACT achieves convergence with significantly less training data.

the performance degradation of the model's selected algorithms compared to the optimal choice. An average slowdown of 1 indicates that the autotuner's selections are optimal, while higher values reflect slower (and thus worse) performance.

We adopt the "convergence criterion" of an average slowdown $\leq$ 1.03 [11]. Convergence represents a predefined standard indicating that the autotuner's selections are sufficiently accurate to conclude the training process.

In Fig. 3, we compare the previous autotuners using the average slowdown metric and simulated datasets from [11]. In this experiment we iteratively train the autotuners with progressively smaller training datasets and monitor the average slowdown. When the average slowdown exceeds the convergence criterion, it signifies that the autotuner's performance is no longer "good enough." The x-axis represents the amount of training data as a percentage of the possible training points, decreasing from left to right. The FACT methodology achieves an average slowdown below the convergence criterion with significantly less data than required by [3]. We use FACT as our point of comparison in this work.

III. Training Point Selection

A. Challenge

FACT's main bottleneck lies in the training data collection process. To select training points, FACT utilizes DeepHyper [19], a tool that trains a surrogate model, in other words, an additional model aimed at understanding collective performance. During the iterative training process, FACT queries DeepHyper for benchmark results. DeepHyper identifies the point that will most enhance the surrogate model's comprehension, benchmarks it, and reports the findings. FACT then uses this data to train its ML model, which is ultimately employed for algorithm selection.

FACT adopts an indirect approach because its ML model, a random forest regressor, lacks a straightforward method for selecting training points. Consequently, FACT's training point selections are tailored to the target environment but not specifically to its ML model, leading to suboptimal choices. Moreover, FACT requires an entire additional concurrent application (DeepHyper) and trains two ML models. We address these inefficiencies and eliminate the need for DeepHyper. We implement a custom point selection methodology for the random forest that yields better selections.

B. Improvements

We use jackknife variance calculations to derive training points directly from ACCLAiM's random forest regressor.

Jackknife is a statistical technique that calculates summary values through resampling [12]. Suppose we have n values $p = (p_1, p_2, \ldots, p_n)$, where p_i is the ith value. Let x_p equal the mean of p.

We calculate jackknife by creating *jackknife samples*. There are n jackknife samples. The i^{th} jackknife sample equals the mean of p with p_i removed. By removing single values, we generate n unique samples. Let x be the means of the jackknife samples, where $x = (x_1, x_2, \ldots, x_n)$ and x_i is the mean of p with p_i removed. Then the variance (σ^2) is calculated as

$$\sigma^2 = \frac{\sum_{i=1}^{n} (x_p - x_i)^2}{n-1}.$$

We select the jackknife technique because of its robustness and fixed computational cost. This computation is embarrassingly parallel across candidate points and completes in under one second per iteration on a single CPU core. It allows us to repeatedly recalculate variance during active learning without adding significant overhead. The dominant cost remains the microbenchmark execution.

We apply the jackknife technique to the predictions of the decision trees within the random forest model, which was first proposed by [20]. Random forest is an *ensemble* ML model, meaning its predictions are the average of individual ML models, in this case, decision trees. We map the random forest to the jackknife technique as follows:

1) Let $n =$ the number of decision trees in the random forest.
2) Let $p =$ the set of predictions from the decision trees; p_i is the prediction from the i^{th} decision tree in the forest.
3) Let $x_p =$ the mean of p.
4) Calculate every x_i by removing each p_i one at a time.
5) Input x_p and x_i into the jackknife variance equation.

For each potential training point, we repeat the jackknife calculation and select the point with the highest variance as the next training point to collect. By prioritizing high-variance points, we effectively supply the ML model with data that addresses gaps in its knowledge, thereby reducing the number of points needed to develop a robust model.

The impact of ACCLAiM's improved training point selection methodology is shown in Fig. 4, which plots training time vs. average slowdown for simulated *MPI_Allgather*. We calculate training time by summing the benchmark execution times. The goal for the training methodologies in this experiment is to decrease their average slowdown and converge as quickly as possible. We mark when each training methodology reaches the convergence criterion, previously adopted as 1.03.

ACCLAiM converges in 2.3x less time than the previous state of the art (FACT). ACCLAiM's model-specific selections create a significant reduction in training data collection time. We note that these measurements include only the benchmarking

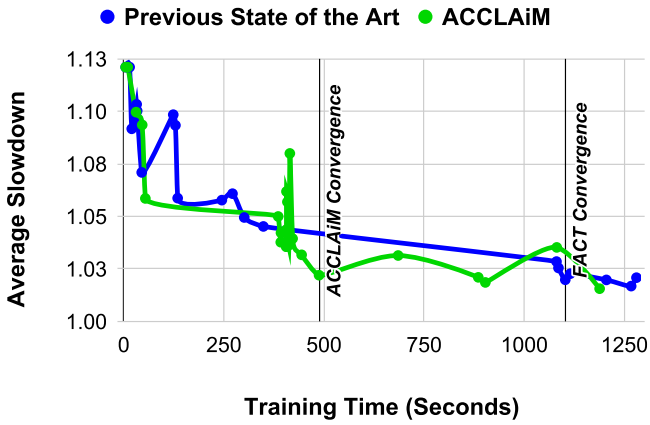


Fig. 4. Comparison of training data collection time using ACCLAiM's training point selection methodology vs. the previous state of the art (simulated *MPI_Allgather*). By intelligently selecting points based on its model's variance, ACCLAiM converges in less than half the time.

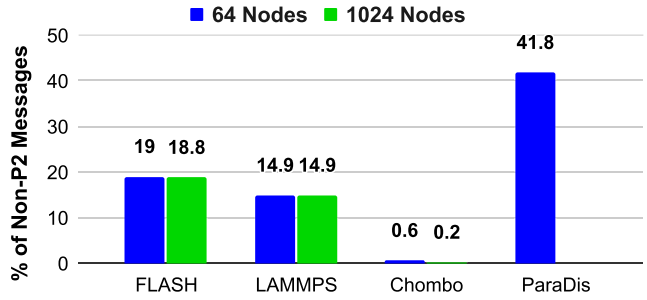


Fig. 5. Percentage of message sizes that are non-power-of-two sizes in HPC applications (1,024-node trace data is unavailable on ParaDis). 15.7% of collective calls use non-power-of-two message sizes, so we must consider their performance.

overhead. They do not account for the additional acceleration from simplifying the point selection process by eliminating the secondary (surrogate) ML model. In practice, we expect even greater speedups.

IV. Non-Power-of-Two Points

A. Challenge

FACT operates under the assumption that all feature values are power-of-two (P2) values. Introducing non-power-of-two (non-P2) values significantly expands the search space for training, necessitating a larger number of training points and substantially increasing the overall data collection time. In simulation, non-P2 values can easily be avoided; but in production, any feature values may be used by applications.

"Number of nodes" and "processes per node" generally follow a particular pattern for each system based on its architecture. For example, jobs on Aurora most commonly use 12 processes per node and jobs on Polaris most commonly use 4, respectively, one for each GPU. These patterns can be mapped onto the normalization technique for these feature values. For P2 feature values like "processes per node" on Polaris or "number of nodes" on both systems, we use a log_2 normalization as in FACT. Log-scale representations have proven effective for ML models operating on values spanning multiple orders of magnitude [21]. For non-P2 features, we map a subset of values into the normalization. For example, for "processes per node" on Aurora, we consider "processes per node" of 1, 4, 12, 48, and 96, mapping to 1, 2, 3, 4, and 5, respectively, when input into the ML model.

On the other hand, the feature value "message size" is more complex. While messages typically use datatypes such as *char* and *int*, which have P2 B sizes, applications may send a non-P2 count of these datatypes, resulting in an overall non-P2 message size.

To analyze application behavior, we profiled traces from Lawrence Livermore National Laboratory [22]. This dataset, released in 2020, contains communication traces from four

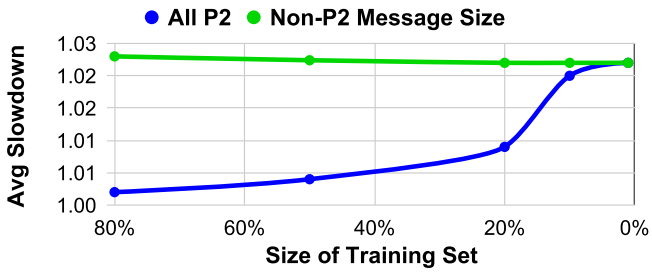


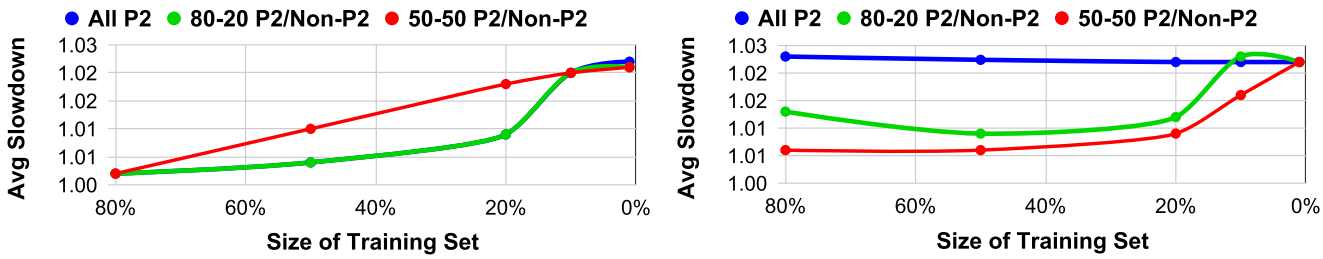
Fig. 6. FACT performance using a non-P2 test set for *MPI_Bcast*. The model, when trained only with P2 values, does not learn the trends of non-P2 message sizes.

production HPC applications (AMG2013, MiniFE, LULESH, and ParaDis) collected on the Stampede2 supercomputer [23] at scales ranging from 64 to 1,024 nodes. The traces capture over 10 million MPI collective calls, making them suitable for analyzing real-world usage patterns of collectives. Fig. 5 reveals that 15.7% of message sizes across the four applications are non-P2. This percentage remains consistent for both small- and large-scale jobs (note that 1,024-node trace data is unavailable for ParaDis). Given that a significant portion of application collective calls involve non-P2 message sizes, it is crucial to address performance for these values.

To evaluate the previous state-of-the-art performance with non-P2 message sizes, we reassessed FACT using a non-P2 test dataset for *MPI_Bcast*. We selected *MPI_Bcast* because two of its algorithms (*binomial* and *scatter_recursive_doubling_allgather*) favor P2 feature values, while the third one (*scatter_ring_allgather*) does not, making it a particularly interesting collective to study.

We collected two new 64-node datasets. One dataset uses all P2 values, consistent with FACT's original evaluation. The other dataset consists solely of randomly selected non-P2 message sizes. We trained FACT (which uses only P2 points for training data) and tested it separately on the new datasets.

The results, depicted in Fig. 6, reveal that the FACT methodology yields an ML model with notably inferior performance for non-P2 message sizes. "Non-P2 Message Size" exhibits a significant average slowdown across the entire graph. The model fails to optimize performance for this test set, even with substantial amounts of training data ($> 80\%$). The underlying



(a) All-P2 Test Dataset (b) Non-P2 Message Size Test Dataset

Fig. 7. Performance of ACCLAiM's P2 training data incorporation for P2 and non-P2 message size test datasets for *MPI_Bcast*. ACCLAiM's 80-20 split maintains P2 performance while dramatically improving non-P2 performance.

issue is that the model cannot learn trends in the data that are specific to non-P2 message sizes. For example, some algorithms partition the message, and non-P2 message sizes may lead to uneven partitions and therefore performance differences that will not appear in P2 data.

To tackle this challenge, we integrate non-P2 message sizes into ACCLAiM's training data selection methodology.

B. Improvements

To incorporate non-P2 training data, ACCLAiM occasionally selects a random non-P2 message size closest to the P2 message size originally selected via the jackknife calculation. For example, if the training point has a message size of 8, we select a new message size between 6 and 12 that is not 8. To determine the frequency of non-P2 data, we use empirical measurements to balance P2 and non-P2 performance.

In Fig. 7 we re-evaluate the "All P2" and "Non-P2 Message Size" test datasets from Fig. 6 with various amounts of non-P2 training data. We consider training sets with all P2 data, a 50-50 data selection split, and an 80-20 split. Each training point includes the same total number of training points, so selecting 50% non-P2 points means that we remove half of the P2 points.

As shown in Fig. 7(b), a 50-50 split maximizes non-P2 performance. However, it sacrifices P2 performance, as shown in Fig. 7(a). We select the 80-20 split because it preserves P2 performance while significantly improving non-P2 performance. By incorporating 20% non-P2 variants, ACCLAiM learns to accurately predict non-P2 points without sacrificing P2 accuracy or additional data collection time.

V. Model Testing

A. Challenge

In each active learning cycle, we assess the ML model's performance for convergence. FACT calculates average slowdown to evaluate convergence, which requires a separate "test set." Previous simulations ignored the time needed to gather test data [3], [10], [11]. In practice, test data is benchmarked similarly to training data but must remain distinct. Although test points consume valuable data collection time, they cannot enhance the model.

FACT has minimized the *training* set size to ∼1% of the feature space. However, the *testing* set still encompasses 20% of the space to ensure a high-quality evaluation. Consequently, test data collection time significantly exceeds training data collection time, inflating the overall collection time.

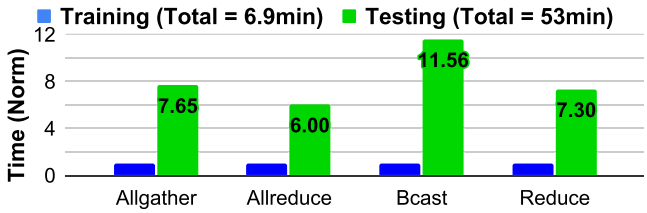


Fig. 8. Training and test dataset collection time for the simulated experiments. Without optimization, test data becomes the bottleneck of the autotuner design.

Fig. 8 illustrates the comparison between the 20% test set and training data collection times, normalized to the training data collection time for each collective. Each collective requires 6–11x more time for test data collection than for training data. FACT's 24-hour training time estimate neglects test data collection, potentially extending machine time to a week or more in reality, which is excessively long. This paper introduces a novel strategy to assess model performance, eliminating the need for a test set entirely.

B. Improvements

In ACCLAiM, we avoid the need for a separate test set by reapplying the jackknife technique from our training point selection methodology. We calculate the cumulative variance across all points to estimate the model's overall uncertainty across the feature space. We find that this summary statistic correlates well with average slowdown, allowing us to detect convergence.

To confirm that cumulative jackknife variance correlates with model performance, we performed a simulated experiment tracking average slowdown and variance. In Fig. 9 we see that variance correlates with average slowdown.

We observe that both metrics follow the same downward trend. At around 400 seconds, the model collects a training point with an unexpected result, spiking the average slowdown as the model overcorrects. The spike in average slowdown is matched with a spike in variance. This result shows that variance is capable of mimicking fine-grained changes in performance and is therefore well suited for convergence detection. We proceed

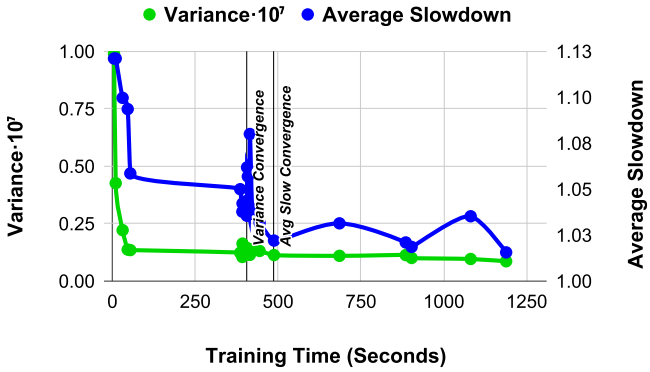


Fig. 9. Cumulative jackknife variance (left axis) and average slowdown (right axis) as a function of training time for ACCLAiM (simulated *MPI_Allgather*). Variance converges similarly to average slowdown, allowing ACCLAiM to detect convergence without a test set.

with cumulative variance as a proxy for average slowdown and therefore our convergence criterion.

VI. Data Collection

A. Challenge

Modern HPC networks use a hierarchical topology, meaning they are structured in various layers or levels, each with varying bandwidth and latency characteristics. To make the correct algorithm selections, we must accurately capture the network dynamics during our training process.

Within a layer, the network may share resources, for example, switches and physical links. Simultaneous communications between separate nodes may cause contention for these resources and negatively impact performance, skewing the autotuner's data. ML autotuners must avoid these issues to accurately measure algorithm performance.

Previous works collect all data points sequentially to avoid causing network congestion issues that alter performance. Although sequential collection is reliable, it is inefficient, especially for large-scale systems. We develop a strategy to safely run microbenchmarks *in parallel*, thereby collecting equivalent training data in a fraction of the time, while still avoiding network congestion.

B. Improvements

To facilitate parallel data collection, we leverage the hierarchical network topology prevalent in modern supercomputers. Fig. 10 illustrates a two-layer dragonfly topology. We apply our technique to dragonfly because of its use in Aurora and Polaris. Nonetheless, our solution is suitable for all hierarchical topologies, including fat trees, making it widely applicable.

In the first layer, the dragonfly group, nodes are connected using a high-radix virtual switch, which may comprise multiple physical switches. The second layer connects the dragonfly groups using the same strategy. This layer typically has a lower bisection bandwidth than the cumulative bandwidth of the groups, for example, 70%. Network congestion will occur when separate applications share network resources within a layer. For example, we cannot allow two benchmark runs within the same dragonfly group.

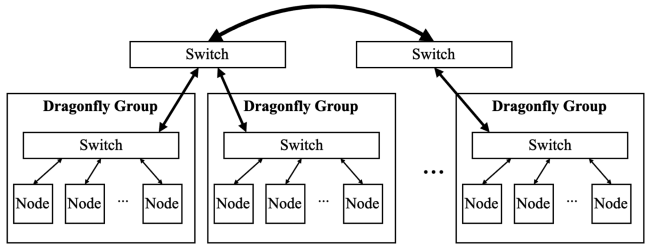


Fig. 10. Simplified two-layer dragonfly topology. Jobs must be scheduled to avoid contention within each layer.

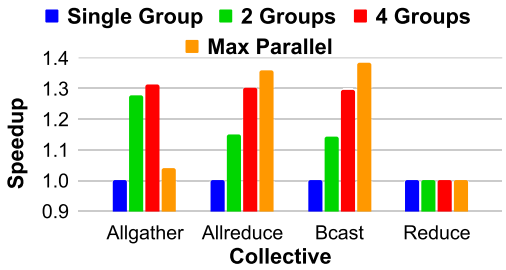
We employ a greedy algorithm to execute benchmarks concurrently across a hierarchical network. Rather than choosing one training point, we create a list of potential training points ranked by variance. Then, we develop a schedule to allocate benchmarks across different nodes for parallel execution. The algorithm operates as follows:

1) Select the next uncollected point p from the list, which requires n nodes.
2) Attempt to schedule p on the next n "unused" nodes.
3) If there are at least n "unused" nodes, the schedule attempt succeeds. Mark those nodes and any additional nodes in the same network layer (e.g., dragonfly group) as "used," and repeat.
4) If there are fewer than n "unused" nodes, the schedule attempt fails. Exit, and run all successfully scheduled benchmarks in parallel.

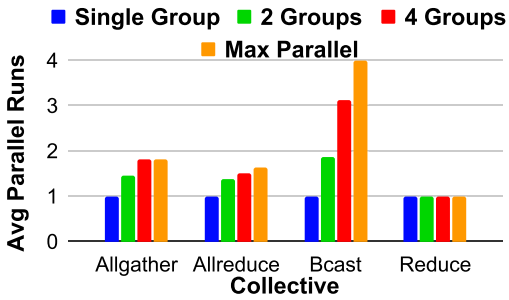
This algorithm avoids network congestion by disallowing different benchmarks to run in the same network layer. Note that it also assigns nodes sequentially; for example, another dragonfly group cannot be used by a single benchmark before the previous one is "used." In networks with multiple network layers like Aurora, which has 8-node chassis subgroups within each dragonfly group, this strategy enables massive parallelism for large-scale jobs. As a result, we are able to train ACCLAiM at larger scales without increasing the training time.

To simulate our parallel data collection strategy, we distribute our simulation dataset for *MPI_Allgather* across four different topologies: all 64 nodes in one group, 32 nodes in two groups, 16 nodes in four groups, and 64 separate groups. We refer to the "separate groups" topology as "Max Parallel." These topologies range from no parallelism (Single Group) to "Max Parallel." Results are shown in Fig. 11. Data collection is accelerated by up to 1.4x by running benchmarks simultaneously. Even in topologies with limited parallelism, we observe significant speedups ($\sim 1.3\times$).

An interesting case appears in Fig. 11(a) for the "Max Parallel" topology. Parallelization speedup decreases compared with other topologies, as "Max Parallel" allows a low-latency benchmark to run alongside a high-latency benchmark that would otherwise be scheduled later. This prevents the high-latency benchmark from being parallelized with the next high-latency benchmark, forcing them to run sequentially. Other topologies prevent the first parallelism opportunity, inadvertently enabling



(a) Speedup



(b) Average number of benchmarks run in parallel

Fig. 11. Average speedup and parallelism exposed by ACCLAiM's parallel data collection (simulation). ACCLAiM achieves a 1–1.4x speedup by running benchmarks in parallel.

the second. This scenario illustrates the limitation of greedy algorithms.

VII. ACCLAIM

We combine all our improvements to create ACCLAiM, our ML autotuner. We describe how ACCLAiM is used, its implementation details, and considerations for expansion.

A. Workflow

Figure 1(b) shows how ACCLAiM is used in a production environment. ACCLAiM is an "allocation-time" tuner, meaning that it trains its ML models once a job is scheduled onto the system just before the application runs.

User Input: The user submits a job to the HPC system and includes the ACCLAiM tuning command in the preamble of the job script. In order to use ACCLAiM, the only knowledge required from the user is which collectives to tune. We expect this information to be common knowledge for highly optimized applications. If not, users should first profile their code to determine which collectives are used and how much time is spent on them overall, to ensure tuning is worthwhile.

Training: Once the job is scheduled, ACCLAiM trains its ML models. For the random forest model, it uses the *RandomForestRegressor* from the *scikit-learn* Python package [24]. ACCLAiM uses the OSU microbenchmark suite v7.5.1 [25] to measure training points using the default benchmark configuration, such as the number of warm-up and measurement iterations. We use the suite's default metric, median latency, as our performance measurement. It uses a single random forest model per collective and enumerates "algorithm" as an additional feature.

TABLE I
RULE CREATION LOGIC

Predictions 1st Point (A), Non-P2 Point (B), 2nd Point (C)	Rules
ALG1, ALG2, ALG2	ALG1 @ $\leq$ A
ALG1, ALG1, ALG2	ALG1 @ $<$ C
ALG1, ALG2, ALG3	ALG1 @ $\leq$ A ALG2 @ $<$ C

Configuration File Generation Once training is complete, ACCLAiM generates a new algorithm selection `.json` file. The `.json` file is a list of logic rules that indicate which algorithm to select. An example rule is `if(`$msg\_size \leq 32$`) {algorithm = binomial}`. The rule set must be "complete," with monotonically relaxing requirements. In other words, every possible input must resolve to a selection, and rules must be sorted from most restrictive to least restrictive.

Numerical comparisons use less than (or equal to) operators, so a valid set of rules is as follows:

- *if*$(msg\_size \leq 32)$ *{algorithm = binomial}*
- *if*$(msg\_size \leq 128)$ *{algorithm = scatter_gather}*
- *if*$(msg\_size = any)$ *{algorithm = binomial}*.

In this example, message sizes less than 32 will use *binomial*, message sizes 33–128 will use *scatter_gather*, and message sizes 129+ will return to *binomial*.

ACCLAiM creates the rule set using the ML model's predictions for every P2 point. It iterates through the predictions and detects when the selection changes. When a selection change is detected, ACCLAiM identifies points A, B, and C. Point A is the last point with the old algorithm selection. Point C is the first point with the new algorithm selection. Point B is the non-P2 point halfway between A and C.

We refer to the selected algorithm at each point as "ALG-(Point)" (e.g., ALG-A is the selection at A). ACCLAiM generates rules based on the change in selected algorithm, as shown in Table I. Three possibilities exist. If ALG-A and ALG-B differ, ACCLAiM creates a rule at A to preserve the correct selection at and below A. If ALG-B and ALG-C differ, ACCLAiM creates a rule at C to ensure the selection for the non-P2 values between A and C is correct. If all three differ, ACCLAiM creates both rules. This strategy captures the autotuner's selection with a minimal rule set while ensuring that non-P2 values may have unique selections.

Application Execution: ACCLAiM passes the new. json file to MPICH using an environment variable. The job script then runs the application and outputs its results.

B. Implementation

ACCLAiM's implementation is open-source and available at https://github.com/pmodels/acclaim. It includes many features to improve ease of use and extensibility.

1) Setup: ACCLAiM offers a streamlined setup process, enabling users to initialize the tool with a single command. The setup process accommodates various system configurations, ensuring portability across different HPC environments.

During setup, users must specify the system for parallel scheduling. ACCLAiM supports several system types:

- `polaris` for Argonne's Polaris system
- `aurora` for Argonne's Aurora system
- `serial` for disabling the parallel scheduler, recommended for small-scale systems where parallel collection is not required
- `local` for testing purposes, allowing an infinite number of parallel microbenchmarks

Additionally, the setup script provides optional arguments to further customize ACCLAiM's functionality:

- `num_initial_points`: Determines the number of data points ACCLAiM should randomly sample at the beginning of exploration.
- `convergence_threshold`: Sets the threshold for cumulative jackknife variance, which consecutive iterations must be below to exit
- `timeout`: Defines the maximum duration in minutes before training should exit, even if the convergence threshold has not been met

`num_initial_points` defaults to 3, and it generally does not impact performance, since training will require many more points. *convergence_threshold* directly impacts convergence time and model accuracy, which we explore in our evaluation. The best value for `timeout` is based on user preference and should scale with job length; for example, longer jobs can afford larger timeouts.

2) Supported Collectives: ACCLAiM supports all regular (i.e., non-vector), blocking collectives in MPICH. The supported collectives include the following:

- `MPI_Allgather`
- `MPI_Allreduce`
- `MPI_Alltoall`
- `MPI_Bcast`
- `MPI_Reduce`
- `MPI_Reduce_scatter`
- `MPI_Reduce_scatter_block`

ACCLAiM also supports tuning the composition collective algorithms in MPICH. Composition algorithms are the topology-aware algorithms in MPICH, meaning they perform separate operations for processes within a node vs. across the network and sit on top of the standard collective algorithms in the software stack. For example, a composition algorithm may combine the node-local values to a single process, perform a "leaders-only" collective between nodes with another algorithm, and distribute the result back to the local processes.

Composition algorithms are critical for collectives such as `MPI_Allreduce`, which we explore in our evaluation.

3) Tuning Commands and Arguments: The autotuning process is facilitated by three commands:

- `gen_config_single`: tunes a single specified collective
- `gen_config_multiple`: tunes a list of specified collectives
- `gen_config_all`: tunes all supported collectives (not recommended because of training-time overhead)

The tuning commands offer several shared arguments to define the tuning scope and parameters:

- *N:* The number of nodes involved in the job.
- *PPN:* The number of processes per node used in the job.
- *MSG_SIZE:* The largest collective message size to tune. A larger size, such as 1048576, is recommended even if the application primarily uses smaller messages. This approach ensures that the selection logic effectively addresses network bandwidth constraints.
- *SAVE_FILE:* The location to store the tuned . *json* file created by ACCLAiM. It is advisable to create a separate directory for these tuning files and incorporate the job ID into the filename to prevent simultaneous jobs from overwriting each other's files.

Users are also required to specify the collective(s) to tune where applicable.

4) Applying the Tuning File: Once the tuning process is complete, users pass the location of the newly generated tuning file to MPICH using an environment variable.

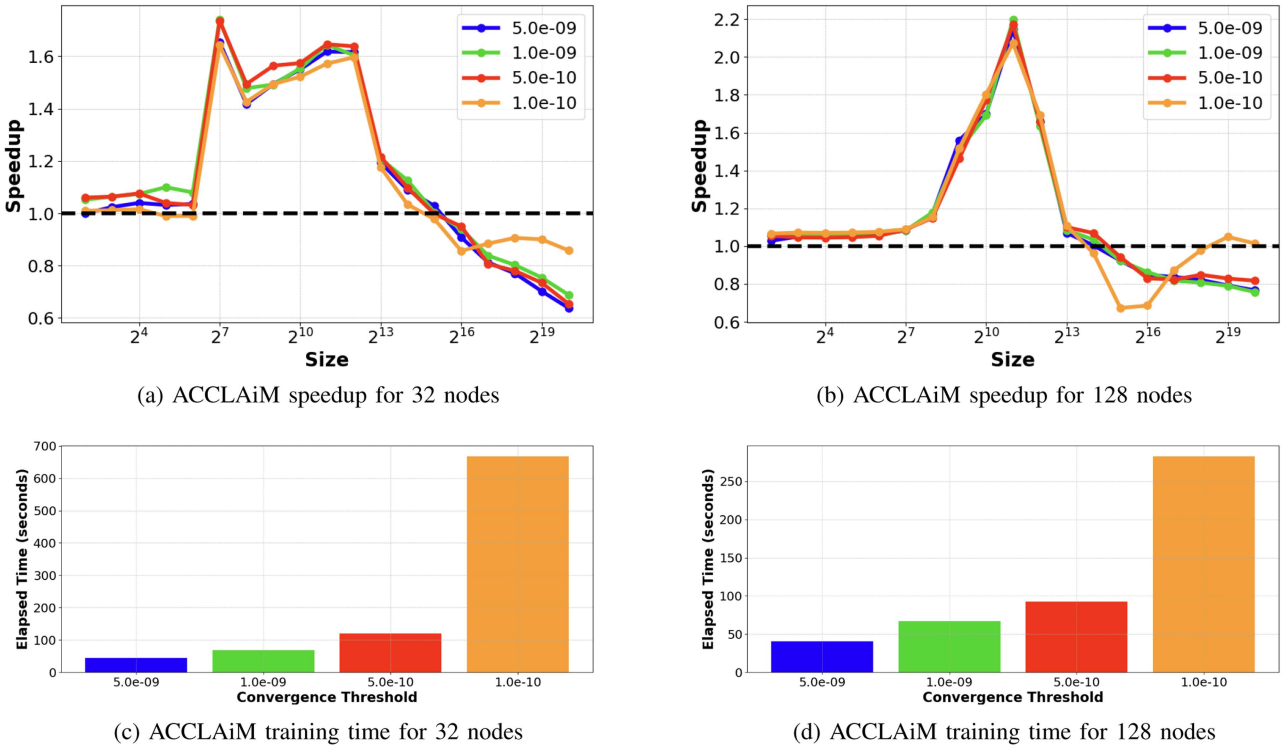
C. Extensibility

ACCLAiM's parallel data collection requires knowledge of the hierarchical network topology. This information is readily available from system documentation or administrators on production HPC systems. ACCLAiM currently supports parallel scheduling on our evaluation systems, Aurora and Polaris. However, its parallel scheduling algorithm is easily extensible to other systems and network topologies. ACCLAiM includes an abstract *topology* class with a set of five simple methods to integrate a new system/topology. To port ACCLAiM to another system, users can add a new derived class and follow the existing examples to understand each method.

D. Topology Requirements

ACCLAiM's parallel data collection requires knowledge of the hierarchical network topology to avoid scheduling concurrent benchmarks that would contend for shared network resources. This information is typically available from system documentation or administrators on production HPC systems. For systems where topology information is unavailable or users are unable to extend ACCLAiM, our implementation provides "serial" mode, which users can specify during setup. Serial mode disables parallel scheduling entirely. This mode is always correct but sacrifices the parallelization speedup (approximately 1.3–1.4$\times$ for smaller scale jobs based on our experiments).

For dynamic network configurations (e.g., adaptive routing that changes during execution), ACCLAiM's impact is limited to the training phase. Since training occurs at job startup before the application runs, the topology observed during training closely matches the topology during application execution within the same job allocation. Cross-job topology variations are addressed by ACCLAiM's per-job retraining design, which captures the specific network conditions of each allocation.



(a) ACCLAiM speedup for 32 nodes

(b) ACCLAiM speedup for 128 nodes

(c) ACCLAiM training time for 32 nodes

(d) ACCLAiM training time for 128 nodes

Fig. 12. ACCLAiM performance analysis for various convergence thresholds using *MPI_Allreduce* on Aurora: (a) and (b) show speedup results, indicating better selections for larger messages with a smaller threshold value of 1×10^{-10}; (c) and (d) show training time, which increases but remains reasonable, making the trade-off for improved accuracy worthwhile.

We note that topology opacity is uncommon on leadership-class systems, where detailed network documentation is typically provided. The extensibility of ACCLAiM (Section VII-C) allows straightforward integration of new topology configurations as they become available.

VIII. Evaluation

In our evaluation, we assess ACCLAiM's capabilities through large-scale experiments on Aurora, with verification experiments on Polaris. Overall, we observe that ACCLAiM provides widespread speedups for multiple collectives while scaling well up to 2,048 nodes on Aurora. We conclude our evaluation with application case studies, showing real-world examples of applications that benefit from ACCLAiM.

A. Evaluation Methodology

In our simulation experiments, we possess performance data for each collective algorithm across all scenarios, enabling us to develop an optimal/oracle selection logic. This data is necessary for calculating *Average Slowdown*, which quantifies the slowdown relative to the oracle.

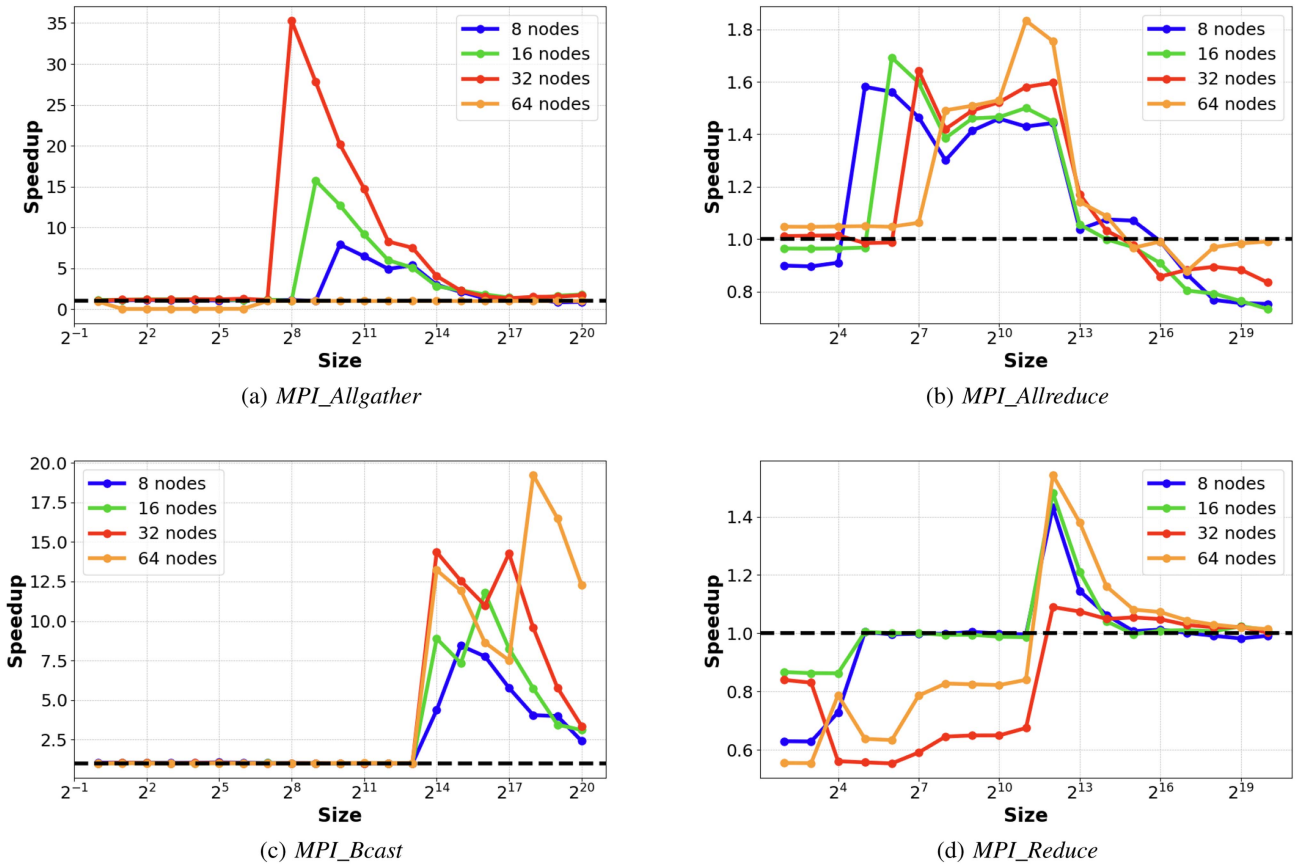
In production, testing every point in the feature set is impractical because of the excessive time required to evaluate every message size and number of processes in a large job. This challenge is the primary motivation for developing ACCLAiM. As a result, we cannot compute statistics such as *Average Slowdown* to assess ACCLAiM's effectiveness. Instead, we measure the speedup of ACCLAiM's selections compared with the default selection logic in MPICH. This approach evaluates the benefits users would gain from ACCLAiM.

We focus on evaluating performance at the maximum job scale. For instance, in a 128-node job, ACCLAiM learns performance across various scales, such as 64 nodes, but we assess its choices only at the full 128 nodes. Tuning all subsets is crucial because large jobs often divide their allocation into multiple communicators and minimize global synchronization/communication when possible. However, full-scale collective operations are common and provide a clear way to test ACCLAiM's scalability. These operations lie at the edge of the feature space, presenting the greatest challenge for ACCLAiM to tune accurately. This approach effectively highlights the tool's strengths and limitations.

We use Aurora as our primary evaluation system. For our Aurora experiments, we use GPU-GPU communication and 12 processes per node, one process per GPU, which is the most common programming model for applications. For experiments using 128 nodes or less, we repeat each experiment 3–5 times within distinct jobs and sets of nodes and average the results. Larger experiments are single runs.

B. Convergence Threshold

We begin by identifying the best convergence threshold for ACCLAiM. As we described in Section V, ACCLAiM uses cumulative variance to detect convergence. With a higher (less



(a) *MPI_Allgather* (b) *MPI_Allreduce*

(c) *MPI_Bcast* (d) *MPI_Reduce*

Fig. 13. ACCLAiM speedup for various collectives on Aurora (higher is better). ACCLAiM generates significant speedup for a wide range of message sizes and scales, while underperforming the defaults in some cases.

stringent) threshold, ACCLAiM will converge earlier but may make poorer selections, while a lower (more stringent) threshold may result in better selections at the cost of increased training time. The goal of these experiments is to identify the convergence value that best balances this trade-off.

We derive the convergence threshold to use in our experiments empirically using a grid search across values ranging from 5×10^{-9} to 1×10^{-11} tuning *MPI_Allreduce*, which is the most popular MPI collective. We retrain ACCLAiM at 32 nodes and 128 nodes with these convergence values to study the trade-off between training time and accuracy.

As shown in the top row of Fig. 12, ACCLAiM struggles to identify algorithms that perform as well as the default for large-message *MPI_Allreduce*.[1] Adjusting the convergence threshold value down to 1×10^{-10} leads to better algorithm selection for the larger messages, significantly improving performance. Although the training time increases, as depicted in the bottom row of the figure, it remains 5–10 minutes, making the trade-off for improved accuracy worthwhile. In our experiments, values less than 1×10^{-10}, for example, 5×10^{-11}, ran into our timeout value of 30 minutes without converging, so they were discarded.

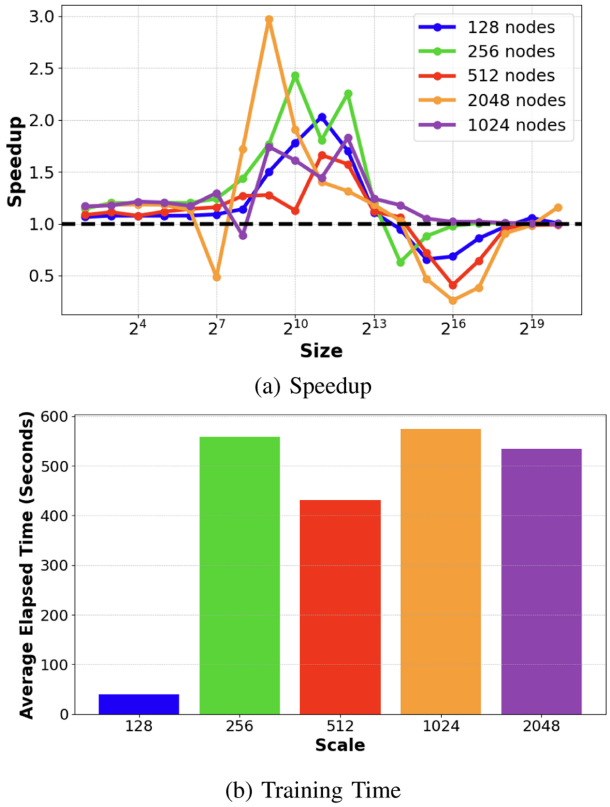
The choice of convergence threshold is crucial for balancing speedup benefits and training time costs. A threshold value of 1×10^{-10} enhances ACCLAiM's ability to optimize performance, especially at larger scales, without excessively prolonging the training process. The training time difference actually decreases for larger scales because ACCLAiM can leverage the increased node count to run more tests in parallel. Based on this result, we proceed to use a convergence value of 1×10^{-10} in our experiments and set it as the default for ACCLAiM.

C. Speedups

We now explore the speedup generated by ACCLAiM's tuning in experiments up to 64 nodes for multiple collectives, including *MPI_Allgather*, *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Reduce*, as shown in Fig. 13. For each collective, we train ACCLAiM for power-of-two number of nodes from 8 nodes to 64 nodes and plot the speedup versus MPICH's default selections for message sizes ranging from 1 B to 1 MB. In these charts, higher is better, as ACCLAiM seeks the maximum speedup over the default selections.

Across all four collectives, we see significant speedups from ACCLAiM. The highlight is *MPI_Allgather*, where we see a

[1] During our testing, we noticed that ACCLAiM can struggle to make optimal selections at the edge of its feature space, i.e., maximum scale and message size. To address this issue, our ACCLAiM prototype can test the default and replace its max-scale selections. We disable this feature for our experiments to fairly represent the capabilities and limitations of ACCLAiM.



(a) Speedup

(b) Training Time

Fig. 14. (a) ACCLAiM speedup up to 2,048 nodes for *MPI_Allreduce* on Aurora, showing how ACCLAiM maintains (and even improves) its advantage at larger scale. (b) Training time stays constant at large scale, demonstrating that ACCLAiM successfully scales on exascale systems.

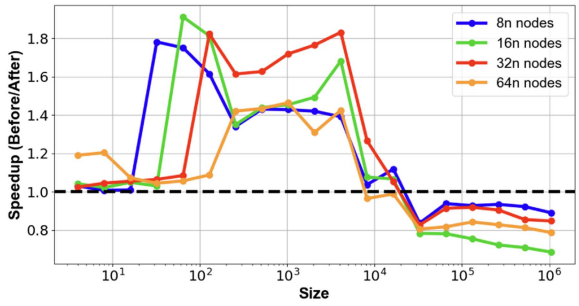


Fig. 15. ACCLAiM speedup for *MPI_Allreduce* on Polaris. ACCLAiM generates a similar speedup to that observed on Aurora, proving its generality across systems.

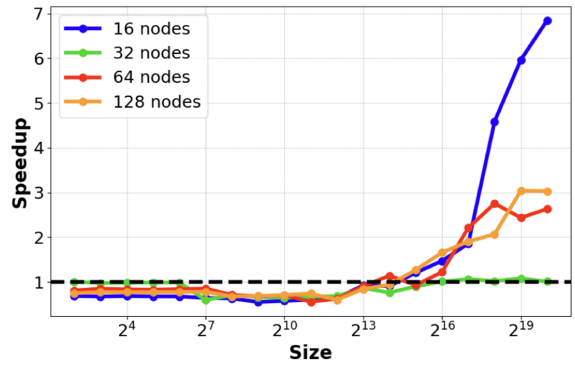


Fig. 16. ACCLAiM speedup for composition algorithm tuning, *MPI_Allreduce*, on Aurora. Composition algorithm tuning creates a large speedup for large message sizes, addressing the weakness observed with the standard tuning in Fig. 13(b).

peak speedup of over 35x. ACCLAiM matches or exceeds the default performance for all messages in *MPI_Allgather*. For *MPI_Allreduce*, ACCLAiM generates speedups for a broad range of message sizes, but it struggles with large message sizes, as we documented in the preceding section. *MPI_Bcast* sees speedups up to 18.3x for large message sizes. The benefit from tuning increases with scale. *MPI_Reduce* observes a slowdown from ACCLAiM's selections for small messages before a positive spike. Studying the selection logic generated by ACCLAiM, we see that the slowdown results from ACCLAiM too eagerly switching from the latency-bound algorithm (*binomial*) to the bandwidth-bound algorithm (*scatter_gather*). We note that *MPI_Reduce* has the fewest implemented algorithms of the collectives we tuned, and ACCLAiM completes the tuning in 1–2 minutes for all scales. We believe this issue could be addressed by further reducing the convergence threshold specifically for *MPI_Reduce*.

D. *Large-Scale* MPI_Allreduce

We continue our scaling experiments for *MPI_Allreduce* up to 2,048 nodes, as shown in Fig. 14(a). The trends from Fig. 13(b) generally continue all the way to the largest scale. The speedup increases up to nearly 3x for small messages around 2^{10} bytes. Importantly, ACCLAiM's training time remains stable, as shown in Fig. 14(b). We observe that the training time remains under 10 minutes (500–600 seconds) for all scales. This result showcases how ACCLAiM's training is tractable even on leadership-class supercomputers.

E. *Polaris*

We provide an additional set of scaling experiments for *MPI_Allreduce* on Polaris to ensure ACCLAiM generalizes to other systems. On Polaris, we use CPU buffers and maximum processes per node, 64, to provide a significantly different scenario compared with our Aurora tests. In Fig. 15 we again see that the performance trends remain similar to 14(b). These results show how ACCLAiM can identify significant speedups on different machines, proving its generality.

F. *Composition Algorithm Tuning*

In Section VII-B2 we discuss how ACCLAiM supports the tuning of composition collective algorithms in MPICH. To demonstrate the benefits of this feature, we conducted a composition algorithm tuning experiment for *MPI_Allreduce* on Aurora, as illustrated in Fig. 16. ACCLAiM achieves significant speedups for *MPI_Allreduce* via composition algorithms, especially for large message sizes. This finding complements our other *MPI_Allreduce* experiments, where ACCLAiM encountered difficulties with large-message *MPI_Allreduce*.

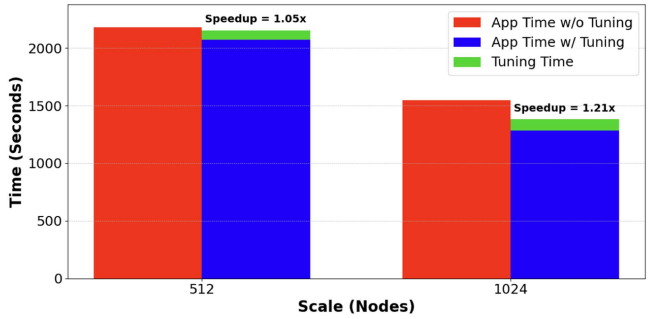


Fig. 17. GAMESS execution time with and without ACCLAiM tuning on Aurora, 512/1024 nodes. The application time without tuning (red) is greater than the application time with tuning (blue) and the ACCLAiM tuning time (green) combined, demonstrating the usefulness of the tool.

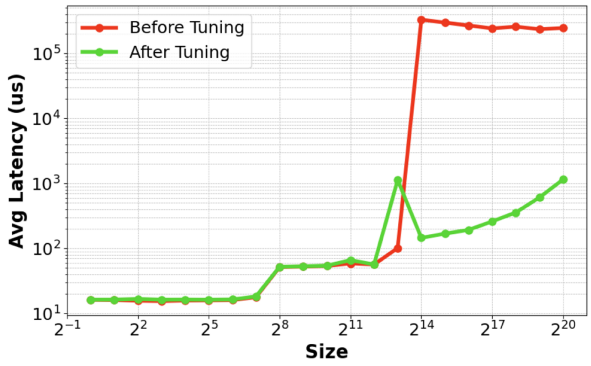


Fig. 18. *MPI_Bcast* performance before and after tuning, 1,024-node GAMESS experiment. ACCLAiM corrects the poor default selection for large message sizes, causing the application speedup in Fig. 17.

The speedups are realized by switching from the default *alpha* composition to *beta* for large messages. The *alpha* composition employs a leader-reduce-broadcast algorithm, where processes on each node reduce their results to a single "leader" process. The leaders then perform an internode *allreduce* using the standard algorithm selection (e.g., *recursive_doubling*, *ring*) and broadcast the result back to the processes on their node. This approach is effective for small messages because of its utilization of low-latency intranode communication. However, it struggles to saturate network bandwidth for large messages during the internode phase, as only a single process communicates for each node.

In contrast, the *beta* composition is a pass-through approach, where the standard algorithm selection proceeds with all processes. This method better utilizes the network, resulting in significant speedups: up to 7x in our experiments, with potential for greater improvements for larger message sizes.

G. Applications

We complete our evaluation with real-world use cases where ACCLAiM aids a production application.

1) Gamess: GAMESS (General Atomic and Molecular Electronic Structure System) is a versatile computational quantum chemistry software package used for performing a wide range of quantum chemical calculations [26]. It enables researchers to study molecular systems by calculating electronic structures, optimizing geometries, and simulating chemical reactions using various methods such as Hartree–Fock, density functional theory, and Møller–Plesset perturbation theory. GAMESS is a critical application on Aurora, widely used for quantum chemical calculations, making it an ideal candidate for evaluating ACCLAiM's impact on real-world workloads.

We study the performance of a sample GAMESS application on two different scales, utilizing 512 and 1,024 nodes, respectively. In a single job we run the application before and after tuning the *MPI_Bcast* collective algorithm selection using ACCLAiM (*MPI_Bcast* is the predominant collective in the application). We assess the impact of ACCLAiM on application runtime and quantify the speedup achieved through this optimization process.

The results, depicted in Fig. 17 , illustrate the application and tuning times for GAMESS. For the 512-node configuration, the application time without tuning was 2179 seconds. Tuning reduced this to 2073 seconds, resulting in a speedup of 1.05x. Similarly, for the 1,024-node setup, the application time decreased from 1550 seconds to 1282 seconds with tuning, yielding a more significant speedup of 1.21x. ACCLAiM required 79 and 99 seconds, respectively, for tuning, far less than the time it saves the application.

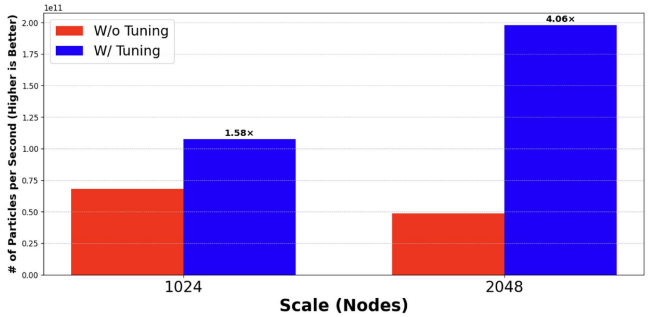
To understand how ACCLAiM improved the performance of GAMESS, we present the performance of *MPI_Bcast* using a microbenchmark before and after tuning in the 1,024-node GAMESS experiment in Fig. 18. The default algorithm selection is a ring-based algorithm, which scales poorly because it has a linear dependency on the number of ranks. ACCLAiM instead selects a doubling algorithm, which scales logarithmically, improving performance by multiple orders of magnitude.

These findings demonstrate the effectiveness of tuning in enhancing the performance of GAMESS, particularly at larger scales. The speedup achieved indicates that tuning can reduce the overall runtime even when accounting for the tuning time.

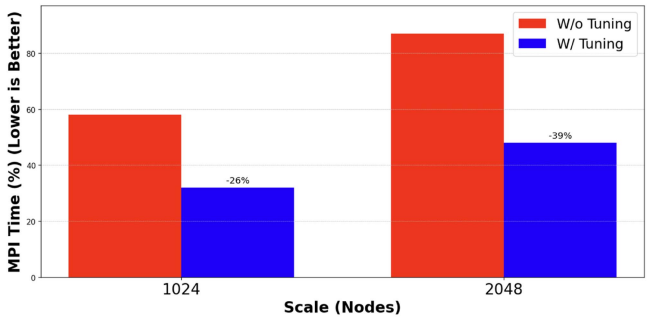
2) SHAMROCK: SHAMROCK is a general-purpose hydrodynamics simulation code focused on astrophysics applications [27]. Originally developed as an exascale-capable implementation of the smoothed particle hydrodynamics (SPH) algorithms from the PHANTOM code [28], SHAMROCK now includes an extensible list of models and solvers for SPH and adaptive mesh refinement including a Godunov model and an N-body fast multipole method.

For our experiments we evaluate SHAMROCK's included weak scaling test. This test is representative of an SPH simulation, where the application synchronizes data using *MPI_Allgather* and/or *MPI_Allgatherv* after each time step. The performance of the test is measured by the number of particles processed per second per step, with the goal of achieving the highest processing rate possible. We experimented with this test at two scales, 1,024 and 2,048 nodes (both 12 processes per node, one per GPU), on Aurora.

To tune SHAMROCK with ACCLAiM, we utilized ACCLAiM's multicollective feature to tune both *MPI_Allgather* and *MPI_Allgatherv*. Despite tuning two collectives, the training



(a) Processing Rate Improvement (Higher is Better).



(b) MPI Fraction of Total Runtime (Lower is Better).

Fig. 19. SHAMROCK with and without ACCLAiM tuning on Aurora, 1,024/2,048 nodes. (a) The application performance increases by $1.6\times$ on 1 024 nodes and by $4.1\times$ on 2 048 nodes. (b) The time spent in MPI communication drops from 58% to 32% on 1 024 nodes and from 87% to 48% on 2 048 nodes. These improvements demonstrate how ACCLAiM accelerates useful work at leadership-class scales.

time remained small: 450 seconds for 1,024 nodes and 267 seconds for 2,048 nodes. At both scales, ACCLAiM successfully recognized that the default algorithm, *ring* for both collectives, was greatly outperformed by the *brucks* algorithm, which is due to *brucks* logarithmic scaling with process count.

Fig. 19 shows the results of ACCLAiM's tuning for SHAMROCK. ACCLAiM greatly accelerates the processing rate of the application by reducing the MPI overhead. At 1,024 nodes, ACCLAiM improves the processing rate $1.6\times$ by reducing the percentage of time spent on MPI from 58% to 32%. The benefits increase on 2,048 nodes, improving the processing rate $4.1\times$ by reducing the MPI time from 87% to 48%.

Typical SHAMROCK simulations run many steps over multiple hours, so they could easily reclaim the ACCLAiM tuning time and experience a performance boost. In summary, ACCLAiM is immediately useful to applications, with the capability to recoup its training time in a single run/job.

IX. Related Work

Collective optimization has been thoroughly explored for over two decades [29], [30], but it remains a continuous challenge to correctly select the optimal algorithms on state-of-the-art networks. Numerous methodologies have been proposed to enhance the selection of collective algorithms, with analytical models being the most prevalent approach [4], [5], [6], [7], [8], [31], [32]. A recent proposal in this area by Luo et al. incorporates hardware-specific information into their model [8]. They develop submodules that represent lower-level components of a collective task, enabling mapping to specific hardware elements. These submodules facilitate the integration of advancements such as accelerators into existing algorithms. Among the concerns associated with analytical models, their primary limitation is the high development cost. In contrast, ML offers a black-box solution with automatic scalability, eliminating the effort required to maintain handcrafted models and analyze new algorithms.

Another methodology involves exhaustive benchmarking. Chaarawi et al. 's OPTO tool optimizes individual scenarios in Open MPI through comprehensive searches [9]. However, tools like OPTO demand extensive data collection time, making them less competitive compared with ML models.

Recent studies have suggested developing collective tuning models across entire machines [33] or even multiple machines [34] using Bayesian optimization. However, these methods are constrained compared with ACCLAiM because they fail to address job-specific variability in algorithm performance, which is an increasingly significant issue in modern HPC systems [35], [36].

Sartori optimizes collective communication using a new approach called "skeletonization," that extracts the program's communication pattern using LLVM [37]. We believe this approach is complementary to ACCLAiM and propose future work where ACCLAiM uses the application skeleton instead of microbenchmarks to measure the effectiveness of various algorithms.

The aforementioned approaches are classified as "offline" autotuners, performing optimization prior to application execution. Faraj et al. introduced STAR-MPI, an "online" autotuner that constructs a statistical model during program execution, dynamically selecting MPI parameters [38]. More recently, Hunold et al. created OMPICollTune [39], which builds a performance model for algorithm performance by probing different algorithms using application collective calls.

Generally, online approaches are avoided because of their complex implementation and runtime overhead. Performance modeling and guideline approaches, which are simpler, also utilize runtime information for dynamic selection [40], [41]. However, these tools are constrained by the models and heuristics that guide them, akin to existing solutions in production MPI libraries.

ML is increasingly recognized as a powerful optimization tool in high-performance computing. Pellegrini et al. employed ML to optimize MPI runtime parameters [42], and Isaila et al. developed an ML model for tuning I/O tasks [43]. Mohammed et al. utilized ML to predict failures in virtualized systems and applications [44], and Zhang et al. applied ML models for scheduling HPC batch jobs [45]. Interestingly, Zhang et al. reversed the typical approach by using collective communication to accelerate ML applications [46]. Our work is complementary to the growing corpus on ML-based optimizations for HPC.

X. Conclusion

This paper introduces ACCLAiM, an ML-based autotuner for collective algorithm selection. ACCLAiM addresses key challenges faced by previous ML collective autotuners: training

point selection, handling non-power-of-two data points, model testing, and data collection, reducing the training time to 5–10 minutes even across thousands of nodes. ACCLAiM is the first autotuner capable of accelerating large-scale applications, even when accounting for training overhead.

ACCLAiM is part of a growing array of collective tuning options. Choosing the right approach involves considerations like offline versus online tuning and machine-wide versus per-job strategies. ACCLAiM stands out because it can be deployed by any system user without special privileges or coordination with administrators. It also adapts to dynamic variables that change between jobs.

We anticipate that retraining the autotuner for every job may not be necessary. Our goal is to integrate ACCLAiM with other methods to create a versatile tuner accommodating various tuning styles for a wider range of end users, such as shorter ($<$ 30 minute) jobs that are unable to recoup ACCLAiM's training time. By combining ACCLAiM's advances in training time reduction with other methods, we envision a hybrid approach that retrains when needed and extrapolates across jobs and systems. We also plan to extend ACCLAiM's capabilities to optimize additional system parameters beyond collective algorithms. ACCLAiM represents the beginning of what ML autotuners can achieve in enhancing MPI application performance.

Acknowledgment

This research used the resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

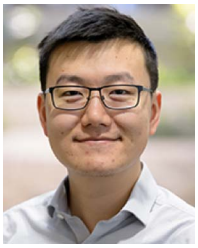
In the preparation for this paper, the AI text generation tool ChatGPT-4o was used for proofreading and light editing for clarity. All text is the original work of the authors.

References

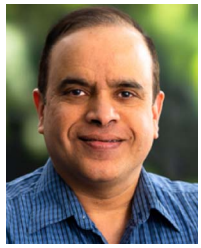
[1] J. Duan et al., "Efficient training of large language models on distributed infrastructures: A survey," 2024, *arXiv:2407.20018*.
[2] H. Khetawat, N. Jain, A. Bhatele, and F. Mueller, "Predicting GPUDirect benefits for HPC workloads," in *Proc. 32nd Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process.*, 2024, pp. 88–97.
[3] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using machine learning," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2020, pp. 259–269.
[4] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically tuned collective communications," in *Proc. SC'00: Proc. ACM/IEEE Conf. Supercomputing*, 2000, pp. 3–3.
[5] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. Dongarra, and E. Jeannot, "Flexible collective communication tuning architecture applied to Open MPI," in *Proc. Euro PVM/MPI*, 2006, p. 26.
[6] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Comput.*, vol. 10, no. 2, pp. 127–143, 2007.
[7] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "MPI collective algorithm selection and quadtree encoding," *Parallel Comput.*, vol. 33, no. 9, pp. 613–623, 2007.
[8] X. Luo et al., "HAN: A hierarchical autotuned collective communication framework," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2020, pp. 23–34.
[9] M. Chaarawi, J. M. Squyres, E. Gabriel, and S. Feki, "A tool for optimizing runtime parameters of open MPI," in *Proc. Eur. Parallel Virtual Mach./Message Passing Interface Users' Group Meeting*, 2008, pp. 210–217.
[10] S. Hunold and A. Carpen-Amarie, "Algorithm selection of MPI collectives using machine learning techniques," in *Proc. IEEE/ACM Perform. Model., Benchmarking Simul. High Perform. Comput. Syst.*, 2018, pp. 45–50.
[11] 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 *Proc. Workshop Exascale*, 2021, pp. 36–45.
[12] B. Efron and C. Stein, "The Jackknife estimate of variance," *Ann. Statist.*, vol. 9, pp. 586–596, 1981.
[13] Aurora, 2025. [Online]. Available: https://docs.alcf.anl.gov/aurora/
[14] Polaris, 2023. [Online]. Available: https://docs.alcf.anl.gov/polaris/
[15] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A high-performance, heterogeneous MPI," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2006, pp. 1–9.
[16] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The MVAPICH project: Evolution and sustainability of an open source production quality MPI library for HPC," in *Proc. Workshop Sustain. Softw. Sci.: Pract. Experiences, held Conjunction Int'l Conf. Supercomputing*, 2013.
[17] MPICH, 1994.Online. Available: https://www.mpich.org
[18] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI usage on a production supercomputer," in *Proc. SC18: Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2018, pp. 386–400.
[19] P. Balaprakash, M. Salim, T. D. Uram, V. Vishwanath, and S. M. Wild, "DeepHyper: Asynchronous hyperparameter search for deep neural networks," in *Proc. IEEE 25th Int. Conf. High Perform. Comput.*, 2018, pp. 42–51.
[20] S. Wager, T. Hastie, and B. Efron, "Confidence intervals for random forests: The Jackknife and the infinitesimal Jackknife," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1625–1651, 2014.
[21] M. Doostmohammadian, M. I. Qureshi, M. H. Khalesi, H. R. Rabiee, and U. A. Khan, "Log-scale quantization in distributed first-order methods: Gradient-based learning from distributed data," *IEEE Trans. Automat. Sci. Eng.*, vol. 22, pp. 10948–10959, 2025.
[22] C. Wang, M. Snir, and K. Mohror, "High performance computing application i/o traces. in lawrence livermore national laboratory (LLNL) open data initiative," 2020. [Online]. Available: http://library.ucsd.edu/dc/object/bb95276921
[23] D. Stanzione et al., "Stampede 2: The evolution of an xsede supercomputer," in *Proc. Pract. Experience Adv. Res. Comput. 2017: Sustainability, Success Impact*, 2017, pp. 1–8.
[24] F. Pedregosa et al., "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[25] "OSU micro-benchmarks," 2002. [Online]. Available: https://mvapich.cse.ohio-state.edu/benchmarks/
[26] G. M. J. Barca et al., "Recent developments in the general atomic and molecular electronic structure system," *J. Chem. Phys.*, vol. 152, no. 15, Apr. 2020, Art. no. 154102, doi: 10.1063/5.0005188.
[27] T. David-Cléris, G. Laibe, and Y. Lapeyre, "The Shamrock code: I–smoothed particle hydrodynamics on GPUs," *Monthly Notices Roy. Astron. Soc.*, vol. 539, no. 1, pp. 1–33, 2025.
[28] D. J. Price et al., "Phantom: A smoothed particle hydrodynamics and magnetohydrodynamics code for astrophysics," *Pub. Astron. Soc. Aust.*, vol. 35, 2018, Art. no. e031.
[29] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High- Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, Spring 2005.
[30] S. Gorlatch, C. Wedler, and C. Lengauer, "Optimization rules for programming with collective operations," in *Proc. 13th Int. Parallel Process. Symp. 10th Symp. Parallel Distrib. Process. IPPS/SPDP 1999.*, 1999, pp. 492–499.
[31] E. Nuriyev and A. Lastovetsky, "Accurate runtime selection of optimal MPI collective algorithms using analytical performance modelling," 2020, *arXiv:2004.11062*.
[32] M. S. Beni, B. Cosenza, and S. Hunold, "MPI collective algorithm selection in the presence of process arrival patterns," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2024, pp. 108–119.
[33] E. Jeannot, P. Lemarinier, G. Mercier, S. Robert-Hayek, and R. Sartori, "Application-agnostic auto-tuning of open MPI collectives using bayesian optimization," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops.*, 2024, pp. 771–781.
[34] M. Han et al., "PML-MPI: A pre-trained ML framework for efficient collective algorithm selection in MPI," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops.*, 2024, pp. 761–770.
[35] A. Khan et al., "An evaluation of the effect of network cost optimization for leadership class supercomputers," in *Proc. SC24: Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2024, pp. 1–16.
[36] M. Salimi Beni, S. Hunold, and B. Cosenza, "Analysis and prediction of performance variability in large-scale computing systems," *J. Supercomputing*, vol. 80, no. 10, pp. 14978–1 5005, 2024.

[37] R. Sartori, "Optimal parameters determination for the execution of MPI applications on parallel architectures," Ph.D. dissertation, Université de Bordeaux, Talence, France, 2024.
[38] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: Self tuned adaptive routines for MPI collective operations," in *Proc. 20th Annu. Int. Conf. Supercomputing*, 2006, pp. 199–208.
[39] S. Hunold and S. Steiner, "OMPICollTune: Autotuning MPI collectives by incremental online learning," in *Proc. IEEE/ACM Int. Workshop Perform. Model., Benchmarking Simul. High Perform. Comput. Syst.*, 2022, pp. 123–128.
[40] S. Hunold and A. Carpen-Amarie, "Autotuning MPI collectives using performance guidelines," in *Proc. Int. Conf. High Perform. Comput. Asia-Pacific Region*, 2018, pp. 64–74.
[41] S. Shudler, Y. Berens, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf, "Engineering algorithms for scalability through continuous validation of performance expectations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1768–1785, Aug. 2019.
[42] S. Pellegrini, J. Wang, T. Fahringer, and H. Moritsch, "Optimizing MPI runtime parameter settings by using machine learning," in *Proc. Eur. Parallel Virtual Mach./Message Passing Interface Users' Group Meeting*, 2009, pp. 196–206.
[43] F. Isaila et al., "Collective i/o tuning using analytical and machine learning models," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 128–137.
[44] B. Mohammed, I. Awan, H. Ugail, and M. Younas, "Failure prediction using machine learning in a virtualised HPC system and application," *Cluster Comput.*, vol. 22, no. 2, pp. 471–485, 2019.
[45] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "RLScheduler: An automated HPC batch job scheduler using reinforcement learning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–15.
[46] B. Zhang, "A collective communication layer for the software stack of Big Data analytics," in *Proc. IEEE Int. Conf. Cloud Eng. Workshop*, 2016, pp. 204–206.

Michael Wilkins Michael Wilkins received the BS degree in computer engineering from Rose-Hulman Institute of Technology and the PhD degree in computer engineering from Northwestern University. He is currently a Maria Goeppert Mayer fellow with Argonne National Laboratory. His research interests include networks and communication for high-performance computing systems, focusing on collective communication autotuning and algorithms. More at mwilkins.org.



Yanfei Guo Yanfei Guo (Member, IEEE) is currently a computer scientist with Argonne National Laboratory and a member of the Programming Models and Runtime Systems group. His research interests include parallel programming models and runtime systems for extreme-scale supercomputing, data-intensive computing, and cloud computing systems. He has contributed to several critical software projects, including MPI (MPICH), Yaksa, and OSHMPI. In recognition of his impactful contributions to the field. He was the recipient of the 2024 ACM Software System Award for his work on the MPICH project, Best Paper Award with the USENIX International Conference on Autonomic Computing (ICAC'13). His research has been widely published in premier peer-reviewed venues such as the ACM/IEEE Supercomputing Conference (SC) and IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (TPDS). He is a member of the ACM.



Rajeev Thakur Rajeev Thakur (Fellow, IEEE) received the PhD degree in computer engineering from Syracuse University. He is currently an argonne distinguished fellow and deputy director with the Data Science and Learning Division, Argonne National Laboratory. His research interests are in high-performance computing, parallel programming models, runtime systems, communication libraries, scalable parallel I/O, and artificial intelligence and machine learning.

Peter Dinda Peter Dinda (Fellow, IEEE) received the BS degree in electrical and computer engineering from the University of Wisconsin and the PhD degree in computer science from Carnegie Mellon University. He is currently a professor of computer science with Northwestern University. His research is on experimental computer systems, particularly parallel and distributed systems, operating systems, and virtualization. More at pdinda.org.

Nikos Hardavellas Nikos Hardavellas (Member, IEEE) received the MS degree in computer science from Carnegie Mellon University, the second MS degree in computer science from the University of Rochester, and the PhD degree in computer science from Carnegie Mellon University in 2009. He is currently a professor of computer science and electrical and computer engineering. He works on parallel systems and computer architecture, primarily on techniques to enable extreme-scale multicore processors, novel memory systems, and practical quantum systems. His research focuses primarily on the quantum system software stack, quantum compilation, fault-tolerant quantum computing and error management, memory-centric computing, and programmable memory systems. His research aims to pave the way to energy-efficient computing by investigating ideas to combat dark silicon, and to speed up the execution of programs by several factors through parallelism extraction, novel architectures, blending of compilers, runtimes, operating systems and hardware, and the use of emerging technologies such as photonics.