ACCLAiM: Advancing the Practicality of MPI Collective Communication Autotuning Using Machine Learning

Michael Wilkins
Northwestern University
wilkins@u.northwestern.edu

Yanfei Guo
Argonne National Laboratory
yguo@anl.gov

Rajeev Thakur
Argonne National Laboratory
thakur@anl.gov

Peter Dinda
Northwestern University
pdinda@northwestern.edu

Nikos Hardavellas
Northwestern University
nikos@northwestern.edu

***Abstract*—MPI collective communication is an omnipresent communication model for high-performance computing (HPC) systems. The performance of a collective operation depends strongly on the algorithm used to implement it. MPI libraries use inaccurate heuristics to select these algorithms, causing applications to suffer unnecessary slowdowns. Machine learning (ML)-based autotuners are a promising alternative. ML autotuners can intelligently select algorithms for individual jobs, resulting in near-optimal performance. However, these approaches currently spend more time training than they save by accelerating applications, rendering them impractical.**

We make the case that ML-based collective algorithm selection autotuners can be made practical and accelerate *production* applications on *large-scale* supercomputers. We identify multiple impracticalities in the existing work, such as inefficient training point selection and ignoring non-power-of-two feature values. We address these issues through variance-based point selection and model testing alongside topology-aware benchmark parallelization. Our approach minimizes training time by eliminating unnecessary training points and maximizing machine utilization.

We incorporate our improvements in a prototype active learning system, ACCLAiM (Advancing Collective Communication (L) Autotuning using Machine Learning). We show that each of ACCLAiM's advancements significantly reduces training time compared with the best existing machine learning approach. Then we apply ACCLAiM on a leadership-class supercomputer and demonstrate the conditions where ACCLAiM can accelerate HPC applications, proving the advantage of ML autotuners in a production setting for the first time.

***Keywords*-MPI, collective communication, autotuning machine learning**

I. Introduction

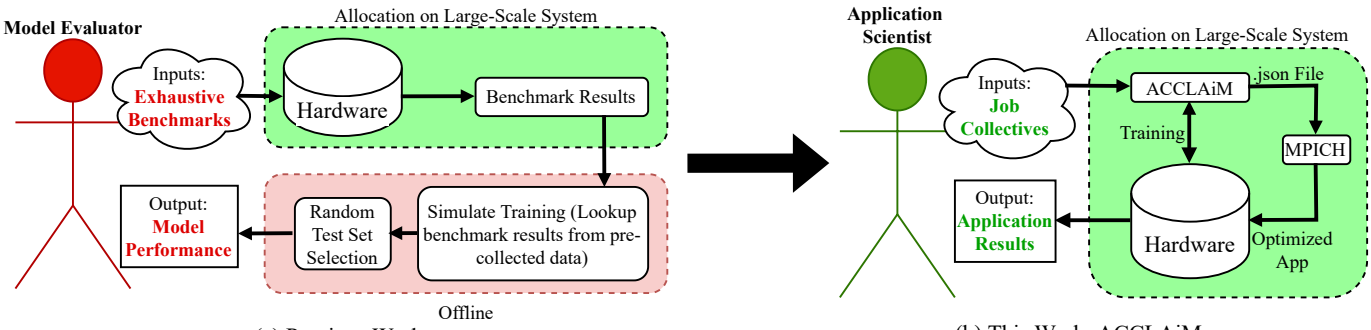
The Message Passing Interface (MPI) is the de facto standard for communication within high-performance computing (HPC) applications. Communication is a substantial bottleneck in modern HPC systems. Many applications running on production HPC systems spend 50%+ of their execution time on MPI rather than actual computation [5]. This percentage is expected to increase in future systems.

Collective operations are among the most popular abstractions in MPI, and they represent more than half of MPI's overhead on production systems [5]. Collectives are a useful and popular abstraction, but their performance varies greatly depending on the algorithm used to implement them. Consequently, MPI libraries sport a growing set of algorithms for each collective. Selecting a suboptimal algorithm can significantly reduce performance (35–40% [11]), but selecting a good algorithm is not easy. The performance of each algorithm is influenced by many factors, from software (e.g., message size) to hardware (e.g., network latency). Dynamic factors (e.g., network congestion, effective topology) vary frequently, making it impossible to create accurate static selections. Because of the complexity of the decision space, developers rely on automated tools to select more performant algorithms.

Multiple methods have been proposed to autotune collective algorithm selection. Examples include analytical models [27], [7], [23], [24], [16] and exhaustive benchmarking tools [4]. Analytical models have failed to gain adoption because they are difficult to implement, maintain, and expand for new algorithms [30]. Production tools such as Intel's MPITune and OPTO [4] use exhaustive benchmarking. Benchmarks must be rerun frequently to account for dynamic factors, making this brute-force strategy impractical for large systems. In practice, it can be used only to tune individual scenarios.

Machine learning (ML) improves upon these ideas by learning to predict algorithm performance patterns [12]. ML collective autotuners use microbenchmarks to measure algorithm performance and then predict the performance for situations they have not evaluated (i.e., "unseen" cases). ML has an inherent advantage over analytical models because it can understand variation caused by factors that are difficult to model analytically, such as real-time and/or machine-specific influences (e.g., network congestion). In addition, predicting algorithm performance for unseen scenarios lessens the benchmarking overhead compared with exhaustive approaches.

Previous work has repeatedly shown that ML autotuners can improve collective performance by up to 30% in simulated



(a) Previous Works (Simulated Evaluation)

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

Fig. 1: Previous works' prototype vs. ACCLAiM. Previous works use offline simulation to show the promise of ML collective autotuners. ACCLAiM aims to make these autotuners practical for applications on large-scale production supercomputers.

experiments [12], [11], [30]. We set out to build the first ML collective autotuner prototype that can be **deployed** on large-scale production supercomputers. Specifically, we target Theta, a supercomputer at Argonne National Laboratory.

While ML autotuners have excellent theoretical results, the state-of-the-art ML design remains overburdened by the amount of training time required [30]. Like exhaustive approaches, ML autotuners must be retrained frequently to account for dynamic influences. On Theta, we must train an ML model at the beginning of every job. In our previous simulations, we estimated a training time of 24 hours for large-scale jobs on Theta [30]. Theta has a maximum job length of 24 hours, so the model will obviously spend more time training than it can save during application execution, rendering it impractical. Additionally, simulations ignore important components, such as non-power-of-two input features, that threaten to further balloon the training time.

In this paper we identify multiple key bottlenecks and issues in the previous state of the art [30] that make ML collective autotuners impractical for large-scale production systems: training data point selection, non-power-of-two points, model testing, and data collection. We address each of these issues through techniques such as jackknife variance calculations [6]. Our approach reduces training time by minimizing the number of training points collected and maximizing hardware utilization. Then we present ACCLAiM, our ML collective autotuner prototype. ACCLAiM is the first ML autotuner prototype that is practical to run on large-scale production systems. In Figures 1 and 2 we showcase ACCLAiM's novelty compared with previous work. Our contributions are as follows:

- A training point selection methodology that uses variance calculations and guided sampling to minimize the required number of points and account for non-power-of-two feature values
- A model testing procedure that uses cumulative variance to avoid collecting any additional data for testing
- Topology-aware parallel data collection to maximize machine utilization during the training process
- ACCLAiM, an ML collective autotuner prototype that is capable of accelerating HPC applications on large-scale production supercomputers.

II. Background

We now introduce our evaluation environments and explain the importance and difficulty of collective algorithm selection and the design and limitations of existing ML approaches. Non-ML solutions are discussed in Section VII.

A. Evaluation Environments

Throughout this work we perform two kinds of experiments. For intermediate results and direct comparison with existing work, we use the testing framework in Figure 1(a) from our previous work [30]. To perform a "benchmark run," we look up the corresponding value in the precollected dataset, which includes exhaustive benchmarking results. We use the same precollected dataset as the previous work to provide the most-level-playing field for comparison. This data was collected by using up to 64 nodes. Each node contains 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.

To evaluate the practicality of our approach in production, we perform experiments using 128 nodes on *Theta*, a leadership-class supercomputer at Argonne National Laboratory. The full machine contains 4,392 nodes, each with an Intel Xeon Phi 7230 with 64 cores and 192 GB of DDR4 memory. The nodes are connected by an Aries Dragonfly network.

In both environments we study the performance of the four most popular collectives from Chunduri et al. [5]: allgather, allreduce, bcast, and reduce. For these collectives, we consider a total of 10 algorithms. Experiments that do not explicitly separate the four collectives show aggregate values.

B. Collective Algorithm Selection

1) Importance: Collective algorithm selection is a vital problem because of its impact on performance. The most popular open source implementations of the MPI standard—Open MPI [10], MVAPICH [19], and MPICH [1]—use heuristics to make selections. This work focuses on MPICH because it serves as the basis of many popular production MPI libraries, including Cray MPI, which is the primary MPI implementation on *Theta*. Hunold et al. [11] found that optimized selections can accelerate collectives by **35–40%** compared with the

default heuristic approach in Open MPI. Because collective operations alone make up a significant portion of HPC application runtimes [5], improving collective performance will, in turn, improve application performance.

2) Difficulty: Optimized collective algorithm selections can accelerate MPI applications, but accurately selecting the best algorithms is difficult because of the large number of influential variables. Two types of variables exist: programmatic and non-programmatic [30]. Programmatic variables are inputs to the collective call. They include message size, number of processes (N), and number of processes per node (PPN). Non-programmatic variables are factors that are invisible to the programmer. Many non-programmatic values exist, including CPU architecture and network topology, latency, and congestion.

To manually select the best collective algorithm for a given scenario, a developer must understand the influences of all programmatic and non-programmatic values. To explain what makes this choice so difficult, we present an example choosing between MPICH's two algorithms for MPI_Reduce on *Theta*. The first algorithm is $binomial$. $Binomial$ constructs a binomial tree where each node reduces its children's values and forwards the result to the "parent." The second algorithm is $scatter\_gather$, which executes in two parts. First, a "scatter" causes each of the n processes to hold the complete reduction for $1/n$ of the values. Then, the root node "gathers" the fully reduced values from each of the other processes.

Conventional logic (based on programmatic values) suggests that $binomial$ is the better algorithm for small message sizes because it requires fewer sequential steps and that $scatter\_gather$ is a wiser choice for large message sizes because scattering maximizes bandwidth utilization.

This traditional argument falters, however, when we consider non-programmatic variables, such as network factors. For example, *Theta*'s scheduler provides no guarantee that nodes allocated to a job will be near each other in the cluster, resulting in higher communication latency between nodes. On *Theta*, we have measured over 2x difference in latency for the same collective algorithm on different jobs and allocations. For networks with high latency, $binomial$ is advantageous even for large message sizes because it uses fewer, larger communications that are less affected by latency. $Scatter\_gather$ wins out in low-latency situations with its many, smaller communications. Note that this argument is just one of many. Other factors, such as network congestion, also alter the effective network latency. There are more variables to consider, but the point is clear: *the design space for collective algorithm selections is far too complex to explore manually.* ML collective autotuners learn patterns in collective algorithm performance, providing optimized collective performance automatically.

3) Dynamicity of Non-Programmatic Values: In simulated experiments, ML autotuners have been shown to understand performance trends affected by a multitude of non-programmatic variables. Since simulations use precollected benchmark data, it is unclear whether these models maintain their accuracy in the face of highly dynamic variables. For example, *Theta*'s "best-effort" scheduler greatly alters algorithm performance for every job. Therefore, an ML autotuner trained during one job may not make accurate predictions for another.

In order to avoid the challenge of dynamic variables, *ML collective autotuners must be retrained at the beginning of every job run on Theta.* This requirement applies the strictest possible standard for training time; each job using the autotuner must be able to recoup the training cost over the course of its single application run. On the bright side, this prerequisite decentralizes our design, greatly improving usability. Instead of trying to coordinate a centralized autotuner across the many users or applications on the system, application scientists can train their own models independently. Any user can utilize ML collective autotuning without special permissions.

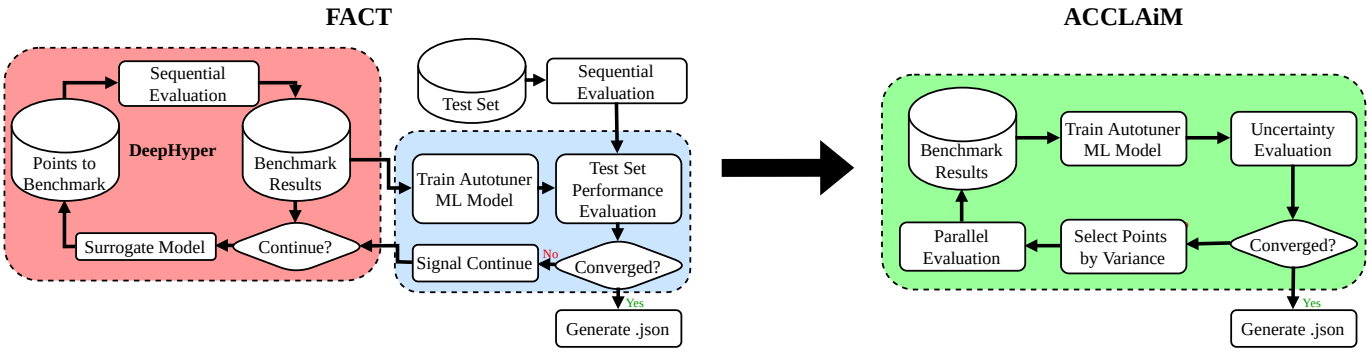
C. Existing ML Autotuner Approaches

1) Baseline Design: In this work we adopt the state-of-the-art ML collective autotuners as our baseline design. Hunold et al. [11] designed the first ML autotuner, which used a single ML model per algorithm per collective. For example, if the MPI implementation had 4 collective operations, each with 3 algorithms, the autotuner would create 12 separate ML models. Each model is responsible for predicting the performance of one algorithm. Each model accepts three inputs: number of processes, number of processes per node, and message size. The inputs are also referred to as "feature values." All possible input values are called the "feature space." Each model outputs a predicted execution time in microseconds.

The autotuner collects training data using microbenchmarks, which in Hunold et al.'s work are a random sample of the feature space. To make an algorithm selection, the autotuner queries the models for each algorithm, and it selects the algorithm of the model with the lowest predicted time. Using precollected data from small clusters (number of nodes ≤ 48), Hunold et al. showed that the autotuner's selections outperformed the default heuristics by 35–40%.

Previously, we recognized that the original training method could not scale for larger machines because benchmarking random points is too costly. To address this challenge, we proposed the FACT methodology that uses active learning to intelligently select training points [30]. Active learning is an iterative process that intelligently selects new training points to more efficiently train the ML model. In simulated experiments FACT maintains near-optimal performance with 6.88x less training data collection time than the Hunold approach. Because of its superior performance, we proceed with FACT as our point of comparison in this work.

Despite its improved performance, FACT introduces multiple new bottlenecks. FACT still requires up to 24 hours of training time for larger-scale (128–512 node) systems, which remains impractical [30]. This time is still dominated by training data collection. Figure 2 illustrates FACT and compares it with our proposed ACCLAiM methodology.



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

Fig. 2: Existing state-of-the-art methodology vs. proposed ACCLAiM methodology. ACCLAiM improves and simplifies the state of the art.

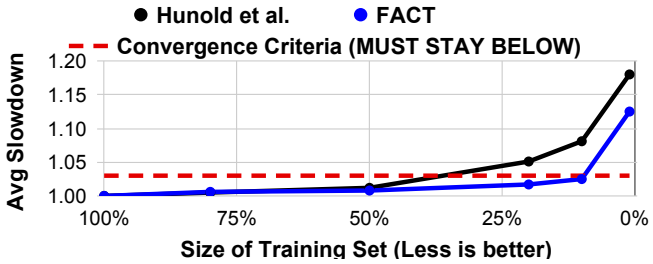


Fig. 3: Performance comparison of previous two state-of-the-art methods (Hunold et al. [11] and FACT [30]). FACT exhibits superior performance, staying below the convergence criteria with far less training data than Hunold et al.'s method needs

2) Evaluation Technique: Throughout this work we frequently adopt our previous performance comparison technique to compare autotuners [30]. For this technique, we use the *average slowdown* metric. *Average slowdown* measures the slowdown of the model's selected algorithms compared with the optimal choice. The best possible *average slowdown* is 1, which signifies that autotuner's selections are completely optimal. A higher value signifies slower (worse) performance.

We adopt the "convergence criteria" of *average slowdown* $\leq$ 1.03 [30]. Convergence is a predefined standard that signifies that an autotuner's selections are accurate enough to end the training process. We repeatedly train the autotuner with increasingly smaller training datasets and record the *average slowdown*. When the *average slowdown* becomes greater than the convergence criteria, we have reached the point where the autotuner's performance is no longer "good enough."

We include our own comparison of the previous two state-of-the-art autotuners using this technique in Figure 3. On the x-axis, we represent the amount of training data as a percentage of the possible training points. The FACT methodology maintains an *average slowdown* below the convergence criteria with far less data than Hunold et al.'s autotuner requires.

III. Existing Challenges

A. Training Point Selection

FACT's primary bottleneck remains training data collection. To select training points, FACT relies on another tool, DeepHyper [3]. DeepHyper trains a surrogate model, which is a separate ML model that also attempts to understand collective performance. During the iterative training process, FACT queries DeepHyper for benchmark results. DeepHyper selects the point that will most improve the surrogate model's understanding. Then DeepHyper benchmarks the point and reports the result. FACT uses the data to train its own ML model, which is eventually used to make algorithm selections.

FACT implements this indirect approach because its own ML model, comprising random forest regressors, does not have a straightforward way to select points to benchmark. FACT's training point selections are specific to the target environment but not to the FACT ML model, resulting in suboptimal selections. In addition, FACT must run an entire additional application (DeepHyper) and train two ML models. We eliminate the inefficiency of a second ML model while producing more accurate selections with a custom point selection methodology.

B. Non-Power-of-Two Points

FACT assumes that all feature values are power-of-two (P2). Incorporating non-power-of-two (non-P2) values greatly increases the search space for training, requiring many more training points and ballooning the overall collection time.

In simulation, non-P2 values can easily be avoided; but in production, any feature values may be used by applications. The "number of nodes" feature value is frequently non-P2 because non-P2 job sizes are common on *Theta*. The job scheduler may prioritize non-P2 node counts to maximize utilization. On the other hand, "processes per node" will rarely be non-P2 for *Theta*, which has 64 hardware threads per node. For systems with non-P2 hardware threads, training can use fractions of their thread count, ignoring P2 versus non-P2.

The last feature value, "message size," is less clear-cut. Messages use datatypes such as *char* and *int*, which have P2 bytes. However, an application may send a non-P2 count of a datatype, making the overall message size non-P2. To study the behavior of applications, we profiled traces from Lawrence Livermore National Laboratory [29]. Figure 4 shows that 15.7% of message sizes across four applications are non-P2. For each application, the percentage is nearly the same for both small- and large-scale jobs (1,024-node trace data

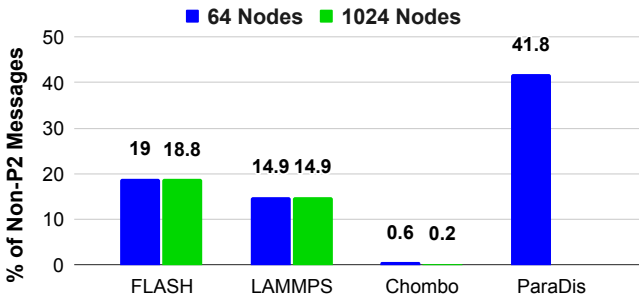


Fig. 4: Percentage of message sizes that are non-power-of-two size in HPC applications (1024-node trace data is unavailable on ParaDis): 15.7% of collective calls use non-power-of-two message sizes, so we must consider their performance.

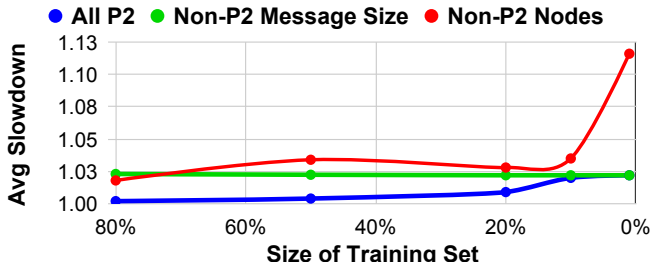


Fig. 5: FACT performance using non-power-of-two test sets for *MPI_Bcast*. The model (trained only with power-of-two values) fails to learn the performance trends in non-power-of-two message sizes.

is unavailable for *ParaDis*). Because a significant portion of application collective calls use non-P2 message sizes, we must address our ML autotuner's performance for these values.

To understand the previous state-of-the-art's performance for non-P2 nodes and messages sizes, we reevaluated FACT using non-P2 test datasets for *MPI_Bcast*. We chose *MPI_Bcast* because two of its algorithms (*binomial*, *scatter_recursive_doubling_allgather*) favor P2 feature values, while the third (*scatter_ring_allgather*) does not, making it the most interesting collective to study here.

We precollected three new test datasets from 64 nodes of a supercomputer with similar hardware used in the evaluation of FACT. One dataset uses all P2 values just like FACT's original evaluation. The other two datasets include only randomly selected non-P2 numbers of nodes and message sizes, respectively. We trained FACT (which uses only P2 points for training data) and tested it separately on all three new test sets.

The results are shown in Figure 5. The FACT methodology produces an ML model with significantly inferior performance for non-P2 test points. For the "All P2" test set, FACT with plentiful training data at 80% performs almost optimally, then slowly deteriorates to the right. "Non-P2 Nodes" has the correct shape, while its *average slowdown* is always higher than the "All P2" results. This trend appears because of the higher performance variability of *MPI_Bcast* algorithms for non-P2 node counts. The ML model is learning the performance patterns at the same rate as the "All P2" set; it just gets punished more severely for incorrect selections.

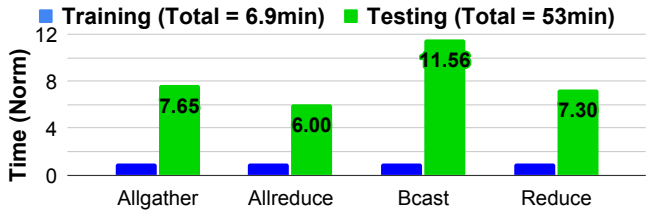


Fig. 6: Training set and test set data collection time for the simulated experiments. Data collection for the test set drastically outpaces ACCLAiM's training data collection time.

"Non-P2 Message Size" shows a significant *average slowdown* across the entire graph. The model does not optimize performance for this test set, even with large (>80%) amounts of training data. The reason is that the model fails to learn the trends in the data. Therefore, we focus on non-P2 message sizes, where the model shows the least ability to learn the performance trends. To address this challenge, we incorporate non-P2 message sizes into our training data selection methodology.

C. Model Testing

In each iteration of active learning, the ML model is measured to check whether the overall performance has "converged." For convergence testing, FACT uses metrics such as *average slowdown* to measure the model's prediction quality. To calculate *average slowdown*, FACT needs additional points to test, aptly called the "test set." Previous simulated experiments ignore the test data collection time [12], [11], [30]. In production, test data points are benchmarked like training points, but the results must be kept separate. Test points consume critical data collection time but cannot be used to train or improve the model.

FACT reduced the required *training* set size to ∼1% of the feature space. However, the *testing* set size needs to cover 20% of the feature space for machine learning methods to work correctly. Consequently, the time to collect test data dwarfs the time to collect training data, drastically inflating the overall data collection time. In Figure 6, we plot the data collection time of a 20% test set compared with the training data collection time. We normalize the values to the training data collection time for each collective. Each collective requires 6–11x more time to collect test data than training data. FACT's 24-hour training time estimate completely ignores test data collection, so this component has the potential to consume a week or more of machine time in production, which is far too large. In this paper we propose a novel strategy to evaluate model performance, eliminating the need for a test set entirely.

D. Data Collection

Previous works collect all data points sequentially. This process is important because HPC networks may route packets indirectly to increase the effective bandwidth between nodes. This policy can create unexpected network congestion between communications, which must be avoided in order to accurately measure collective performance. While safe, a sequential

collection strategy is very inefficient, particularly for larger machines. We avoid network congestion and run benchmarks *in parallel*, up to 100% node utilization, collecting the same training data in significantly less time.

IV. Training Process Improvements

In this section we describe our improvements that address each of the challenges identified in the preceding section.

A. Training Point Selection Improvements

As described in Section III-A, the FACT approach generates training point selections using a second, separate ML model. To simplify the process, we use jackknife variance calculations to derive training points from a single ML model.

The jackknife technique calculates summary values through resampling [6]. Suppose we wish to find the variance of n values. The values are $p = (p_1, p_2, ..., p_n)$, where p_i is the ith value. Let x_p equal the mean of p. We perform a jackknife by creating *jackknife samples*. For a jackknife with n values, 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 $x = (x_1, x_2, ..., x_n)$ 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 use the jackknife technique to calculate the variance of potential training points. We adopt the use of random forest regressors [30]. Applying the jackknife technique to a random forest regressor was first proposed by Wager et al. [28]. Random forest is an *ensemble* machine learning model, meaning its predictions are the average of an ensemble of individual ML models. Random forests consist of decision trees. We use 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 x_i by removing each p_i one a time.
5) Input x_p and x_i into the jackknife variance equation.

We repeat this process for every possible training point. Then we select the point with highest variance as the next training point. By selecting points with high variance, we are providing the ML model with data that fills gaps in its understanding, minimizing the number of points required to train a well-formed model. We include P2 feature values only when using jackknife to limit the number of calculations.

B. Non-Power-of-Two Point Improvements

To address non-P2 points from Section III-B, we incorporate an extra step in the training point selection. Every fifth training point, instead of choosing the exact point with the highest variance, we instead select a point with a random non-P2 message size where the selected message size is the closest P2 value. For example, if the highest variance point has a message size equal to 8, we would select a new message size between 6 and 12 that is not 8. We do so because we experimentally determined that it best balances P2 and non-P2 performance (Section VI-B). By incorporating non-P2 variants, the ML model learns to accurately predict non-P2 points without additional data collection time.

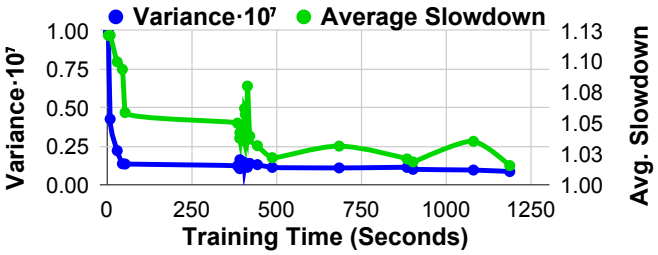


Fig. 7: Variance and *average slowdown* as a function of training time for ACCLAiM. *Average slowdown* converges as variance converges.

C. Model Testing Improvements

As described in Section III-C, previous approaches calculate metrics using a costly set of test points. We wish to estimate model performance *without a test set*, which means we cannot calculate any performance metrics. We instead need a quantity that is measurable during training with minimal overhead and correlates with model performance. To achieve this, we reapply the jackknife technique. We sum the variance from every point to form the correlated quantity.

To test whether this variance measure correlates with model performance, we performed a simulated experiment tracking both the traditional convergence metric (*average slowdown*) and variance. In Figure 7 we see that variance correlates with *Average Slowdown*. We observe that both metrics trend downward over the same time interval. At around 400 seconds, the spike in *average slowdown* is matched with a spike in variance. This event tells us that variance is capable of mimicking fine-grain changes in performance. With these observations in hand, we can proceed with cumulative variance as a proxy for *average slowdown* and therefore our convergence criterion. We provide further evaluation of this technique in Section VI-C.

D. Data Collection Improvements

As described in Section III-D, previous approaches collect training data points *sequentially* to avoid potential network congestion. To enable *parallel* data collection, we leverage the network topology of our target hardware system.

Figure 8 shows a simplified version of the topology. Nodes are numbered sequentially within a rack and across racks. The network has three layers. The first layer connects nodes within a rack. Layer two pairs every two racks. The third layer connects the rack pairs. Network congestion will occur if benchmarks share a single instance of a layer. For example, we cannot allow two benchmark runs on nodes in a single rack. Similarly, if a run is using nodes on both rack 0 and rack 1, we cannot use any remaining nodes on those racks.

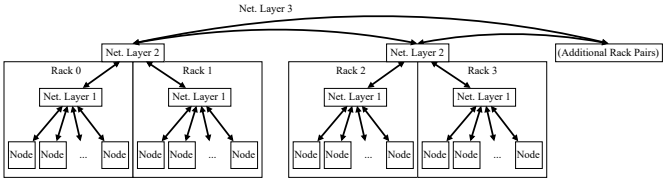


Fig. 8: Simplified topology for our target network (Aries Dragonfly). Jobs must be scheduled to minimize network congestion in the first two layers.

We propose a greedy algorithm to run many benchmarks in parallel. Instead of selecting a single training point, we generate a list of potential training points sorted by variance. Then we generate a "schedule," assigning benchmarks to run on disparate nodes in parallel. The algorithm works as follows:

1) Select the highest-variance uncollected point p, which requires n nodes.
2) Attempt to schedule p on the next n "unused" sequential nodes.
3) If n can fit in the eligible nodes, schedule p on the next n "unused" nodes, mark those nodes and any remaining nodes in the same racks as "used," and repeat.
4) If n cannot fit, exit and run all scheduled benchmarks in parallel.

This algorithm avoids network congestion by disallowing different benchmarks to run in the same rack and scheduling on sequential nodes. By disallowing shared racks, we prevent congestion on the first network layer. Scheduling on sequential nodes prevents congestion on the second network layer because multi-rack benchmark runs cannot simultaneously schedule across the same racks. If a run needs nodes on another rack, it must first fill its original rack, preventing a second run from also being scheduled across those racks.

If two runs schedule across multiple rack pairs and the first run ends within the same pair as the second run begins, the third network layer may see slight congestion. However, because the third network layer is implemented using direct, high-bandwidth connections, we expect incidental congestion to be relatively low. Also, because *Theta* is an active production environment, congestion in the third layer is expected from other applications. We account for third-layer congestion by measuring each collected point multiple times.

This solution is specific to the Dragonfly topology used by *Theta*. Parallel data collection on non-Dragonfly supercomputers may require methodology tweaks for the new environment.

V. ACCLAiM System

To evaluate our improvements, we create ACCLAiM, our ML autotuner prototype. Here we describe how a user interacts with ACCLAiM and the implementation details. ACCLAiM uses an "offline" ML approach, meaning that we train our ML models prior to application execution. Because we retrain the model for every job, however, training consumes execution time during a job's allocation. Figure 1(b) shows how ACCLAiM is designed to be used in a production environment.

User Input: The user submits a job to the HPC system through ACCLAiM. The only additional information required is a list of collectives predominantly used in the application. We expect this collective list to be common knowledge for highly optimized applications. If not, users can provide a conservative list including every collective that *might* require autotuning or use a tool such as Intel's APS [14].

Training: When a job is scheduled in the production environment, instead of immediately running the application, ACCLAiM trains ML models for the specified collectives. To collect training points, we use the Ohio State University microbenchmark suite [2]. For our random forest model, we use the *RandomForestRegressor* from the *scikit-learn* Python package [20]. We use a single random forest model per collective and enumerate "algorithm" as an additional feature.

Configuration File Generation: Once the models have converged, we generate an edited version of the default algorithm selection `.json` file with our models' selections. The `.json` file is a list of logic rules that indicate which algorithm to select. An example rule is `if`$(message\_size \leq 32)$ {`algorithm = binomial`}. The rule set must be "complete," meaning that every possible input must resolve to a selection. The rules also must be pruned such that no two consecutive rules resolve to the same prediction. This step minimizes the selection delay during execution.

To create a list of rules, we collect the ML model's algorithm selections for every P2 point. Naively, we could iterate through the selections and create a rule every time the selection value changes. However, this method abandons the model's non-P2 point selections. Instead, we iterate through the ML model's algorithm selections and detect when the selection changes. When there is a change in algorithm selection between two points A and C with A $<$ C, we requery the model for a selection at B. 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 as "ALG-(Point)" (e.g., ALG-A is the selection at A).

We generate rules based on the change in selected algorithm, as shown in Figure 9. We create three rules: all values below A (inclusive) use ALG-A, all values between A and C (exclusive) use ALG-B, and all values above C (inclusive) use ALG-C. These rules enable unique selections for non-P2 points between A and C. Then, to optimize the rule set and minimize selection delay, we prune these rules. If ALG-A = ALG-B, we can merge the first two rules into a single rule for all values less than C. If ALG-B = ALG-C, we can merge the last two rules into a single rule for all values greater than A.

Application Execution: We direct MPICH to the new `.json` file using an environment variable. Then we run the application and output its results, completing the job.

Neither of the previous state-of-the-art works created a prototype that combines the ML autotuning with job execution. ACCLAiM, on the other hand, completes the entire process while remaining transparent to the user.

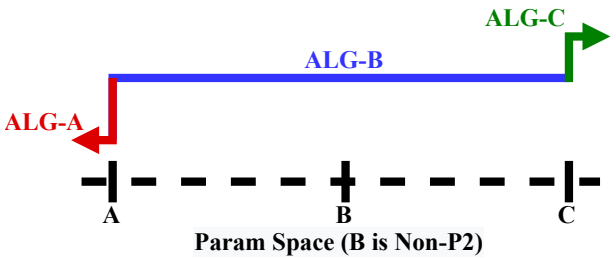


Fig. 9: Rule creation logic. We generate new rules in the configuration file to communicate the ML model's selections to the MPICH library. Each color represents a different rule.

VI. Evaluation

We first compare ACCLAiM with FACT to analyze our individual improvements. For all comparative results, we perform simulated experiments as described in Section II-A to provide the fairest comparison. Then we show ACCLAiM's capabilities on a leadership-class supercomputer, *Theta*.

A. Training Point Selection: Up to 2.3x Faster

We begin with the reduction in training point collection time created by our jackknife-based training point selection methodology. We compare our selection approach vs. FACT's previous state of the art. Figure 10 shows *average slowdown* vs. training time graphs for each collective. For the x-axis, we calculate training data collection time by summing the benchmark execution times. Both training methodologies want to decrease their *average slowdown* as quickly as possible. We mark when each training methodology reaches the convergence criterion, which is the point where we accept the model performance as "good enough" to stop training. We adopt a convergence criterion of *Average Slowdown* < 1.03 [30].

ACCLAiM converges for all four collectives in up to 2.3x less time than the previous state of the art (FACT). *MPI_Allgather* is the most expensive collective to tune and also ACCLAiM's biggest win at 2.3x. The previous state of the art performs slightly better for *MPI_Allreduce* and *MPI_Bcast*, by 1.37x and 1.46x, respectively. Both methodologies converge almost instantly for *MPI_Reduce*. Overall, ACCLAiM's model-specific selections create a significant reduction in training data collection time. The reported speedups 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.

For both Figure 10 and Figure 12 (which we explore in Section VI-C), a few visual oddities are important to understand. First, all lines do not begin 0 on the x-axis. To evaluate model performance, we must collect and supply at least one point for training. If the first training point takes a significant amount of time to collect, a "gap" may appear on the left side of the graph. This behavior is normal and expected; it just indicates that the first training point was expensive to collect. Additionally, large flat sections appear in some graphs. Note that these graphs are discrete, since there cannot be partially collected data points. The lines between points are included to indicate trends. If a point takes a long time to collect and does not greatly affect the model's understanding, a flat section appears on the graph. However, these sections do not represent potential convergence, just expensive point collection.

B. Non-Power-of-Two Points: Now Modeled

Next we evaluate the effects of incorporating non-P2 points in our sampling process. We re-evaluate the "All P2" and "Non-P2 Message Size" test datasets from Figure 5 incorporating various amounts of non-P2 training data.

The goal is to maintain an *average slowdown* as close to 1.0 as possible with minimal training data. We consider training sets with all P2 data, a 50-50 selection split, and ACCLAiM's 80-20 data selection split. Each training point includes the same total number of training points. Selecting 50% of the non-P2 points means that we remove half of the P2 points.

From Figure 11(b), a 50-50 split maximizes non-P2 performance. However, it sacrifices P2 performance, as shown in Figure 11(a). ACCLAiM's 80-20 split preserves P2 performance while significantly improving non-P2 performance. By selecting every fifth point as non-P2, ACCLAiM maintains the "Goldilocks" performance balance.

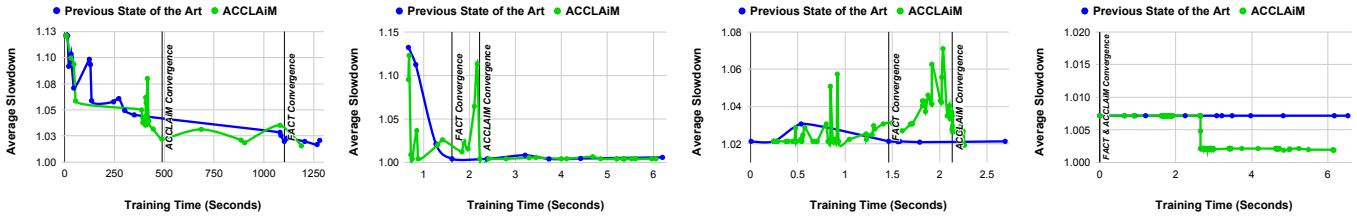
C. Model Testing: Avoid 6–11x Test Set Collection Slowdown

Here we compare cumulative variance as a convergence criterion vs. *average slowdown*. Our variance convergence criterion is that four consecutive training iterations must have a difference in variance less than 10^{-9}. We selected this criterion from prior tuning experience. It works well for both simulation and the production system (discussed in the next section).

In Figure 12 we graph the cumulative variance values on the left vertical axis and *average slowdown* from Figure 10 on the right vertical axis. We mark when both metrics meet their respective convergence criterion. An ideal variance convergence would occur at the same time as the *average slowdown* convergence because the goal is to precisely model *average slowdown*. We accept variance convergences close to the *average slowdown* convergence point if both points produce ML models of nearly equal performance.

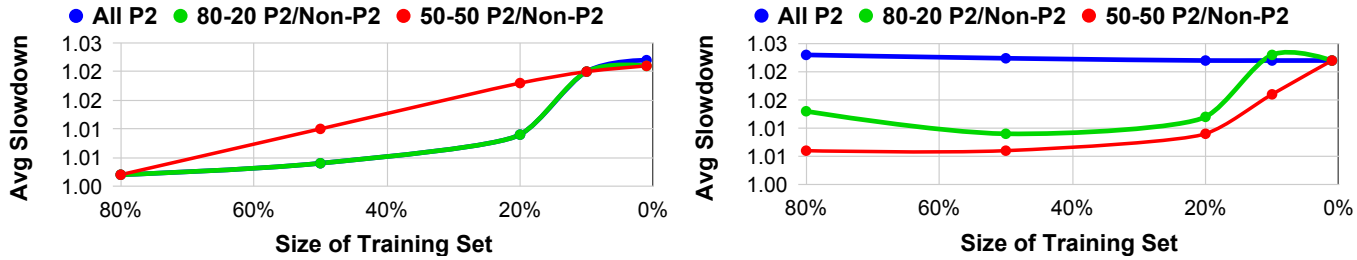
For all collectives, Figure 12 shows that the variance convergence criterion consistently produces trained models with low *Average Slowdown*. *MPI_Allreduce* and *MPI_Reduce* have variance convergence points after the original *average slowdown* points, adding an extra 1.007x to the cumulative training time for all collectives. However, the overall training time is actually reduced by 1.19x because of the other two collectives.

For *MPI_Allgather* and *MPI_Bcast*, the variance convergence point is before the average slowdown point. In both cases, the model-tested variance has an *average slowdown* of 1.04. While this value is above the *average slowdown* convergence criterion, we believe this slight uncertainty is well worth the trade-off of eliminating the testing set. Overall, we avoid the potential 6–11x slowdown from Figure 6 without sacrificing convergence accuracy.



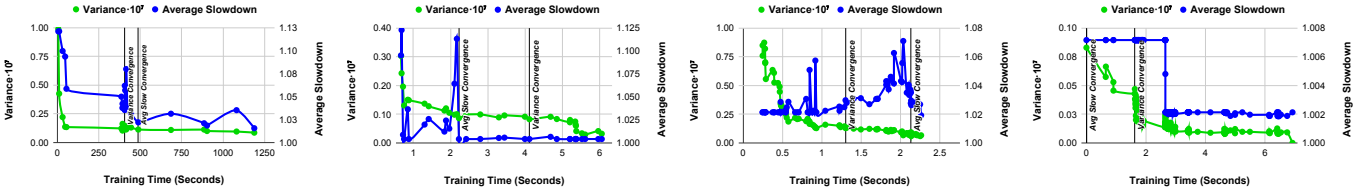
(a) *MPI_Allgather* (b) *MPI_Allreduce* (c) *MPI_Bcast* (d) *MPI_Reduce*

Fig. 10: Comparison of training data collection time using ACCLAiM's training point selection methodology and the previous state of the art for the most popular collectives. Cumulatively, ACCLAiM converges in 2.25x less time.



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

Fig. 11: 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.



(a) *MPI_Allgather* (b) *MPI_Allreduce* (c) *MPI_Bcast* (d) *MPI_Reduce*

Fig. 12: Comparison of convergence time using ACCLAiM's cumulative variance and average slowdown. Using cumulative variance as a proxy for *average slowdown*, ACCLAiM detects convergence 1.19x faster while avoiding a potential 6–11x slowdown caused by test set data collection.

D. Data Collection: 1.4x Faster Using Parallelism

To evaluate our parallel data collection strategy, we schedule the simulation dataset across four different simulated topologies: all 64 nodes on a single rack, 32 nodes each on two racks in a pair, 16 nodes each on four racks in two pairs, and single nodes on different racks all from separate pairs (1-0-1-0...). The "separate pairs" topology represents the maximum parallelism potential, so we henceforth refer to it as "Max Parallel." These topologies represent a range of situations from no/minimal parallelism (Single Rack, Single Rack Pair) to "Max Parallel." The results are shown in Figure 13.

Data collection is accelerated by up to 1.4x by running 1–4 benchmarks simultaneously. Even for topologies with modest parallelism opportunities, we see significant speedups (∼1.3x).

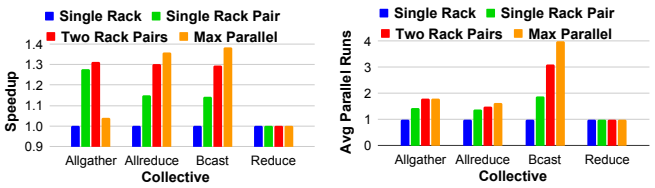
An interesting data point occurs in Figure 13(a) for the "Max Parallel" topology for $MPI\_Allgather$. Here, the parallelization speedup decreases compared with the other topologies. During scheduling, "Max Parallel" enables a low-latency benchmark to run in parallel with a high-latency, succeeding benchmark. The high-latency benchmark is now unable to be parallelized with the next benchmark, which also has a high latency. Running the two high-latency benchmarks sequentially more than eliminates the original advantage. The other topologies disallow the first parallelism opportunity, which coincidentally enables the second. This situation highlights the suboptimal nature of greedy algorithms.

E. Production Practicality: Benefits Typical Jobs on Theta

Now that we have evaluated our contributions individually, we apply ACCLAiM to *Theta*, a leadership-class supercomputer. For these experiments, ACCLAiM selects algorithms up to 128 nodes, 16 processes per node, and 1 MB message size.

The training time is shown in Figure 14. In the larger-scale, production environment, ACCLAiM converges in a matter of minutes, compared with the many hours estimated by the previous state-of-the-art [30]. We cannot make a direct performance comparison because FACT is unable to function in this "real" (not simulated) setting.

We now consider ACCLAiM's impact on application performance. Collective performance is critical to the performance of many HPC applications [27], [5]. Previous works have



(a) Speedup

(b) Average number of benchmarks ran in parallel

Fig. 13: Average speedup and parallelism exposed by ACCLAiM's parallel data collection. ACCLAiM achieves a 1–1.4x speedup by running 1–4 benchmarks in parallel.

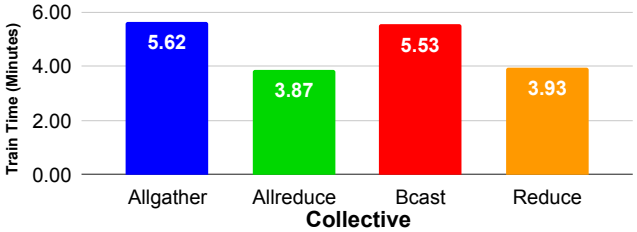


Fig. 14: ACCLAiM training time for up to 128 nodes on a leadership-class supercomputer. By reducing the training time to a number of minutes, we achieve production practicality.

provided examples of how autotuning collective algorithm selections can improve application performance [22], [16]. To assess the broader applicability for our work, we show in Figure 15 the minimum application runtimes required to gain an overall speedup when using ACCLAiM. We present a range of application speedups, which vary depending on the quality of the default selections and the percentage of time the application spends on collective calls. Applications that run for more than a few hours and are slowed even slightly by the default algorithm selections are great candidates for ACCLAiM. For example, an application with a 1.01x speedup from improved algorithm selections only has to run for 6.4–9.5 hours, which is well within the common duration for jobs on *Theta*. By showing that ACCLAiM can accelerate applications with moderate runtimes, we demonstrate ACCLAiM's practicality for large-scale production systems.

VII. Related Work

Collective optimization dates back more than 20 years [26], [9]. Many others have proposed methodologies for selecting collective algorithms besides machine learning. The popular approach is analytical models [27], [7], [24], [23], [16], [18]. The most recent of these proposals is by Luo et al. [16]. They create submodules that represent lower-level portions of a collective task, some of which can be mapped to specific hardware components. Analytical models suffer from other minor concerns, but we believe their ultimate downfall is development cost. Compared with the effort required to maintain handcrafted models and analyze new algorithms, ML provides a black-box solution with automatic expandability.

Another approach is exhaustive benchmarking. Chaarawi et al.'s OPTO tool tunes individual scenarios in Open MPI using

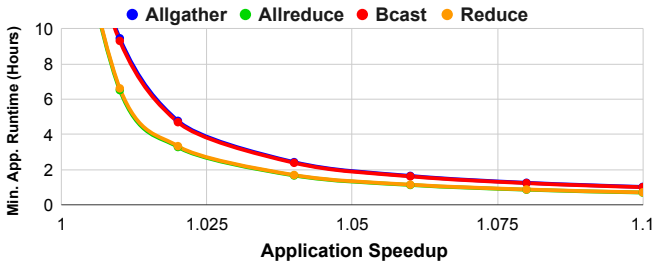


Fig. 15: Minimum application runtime for overall acceleration when using ACCLAiM. In many cases, applications must run for only a few hours to recover ACCLAiM's training time.

a complete search [4]. Tools such as OPTO, however, require far too much data collection time to compete with ML models.

Faraj et al. proposed STAR-MPI, an "online" autotuner that builds a statistical model during program execution and dynamically selects MPI parameters [8]. In general, online approaches are rare because of their complex implementation and runtime overhead. Performance modeling/guideline approaches are simpler and also use runtime information to make selections dynamically [13], [25]. However, these tools are restrained by the models/heuristics that guide them, similar to the existing solutions in production MPI libraries.

Machine learning is becoming a prominent optimization tool across HPC. Pellegrini et al. optimized other MPI runtime parameters using ML [21]. Isaila et al. built an ML model to tune I/O tasks [15]. Mohammed et al. used ML to predict failures in a virtualized system/application [17]. Zhang et al. scheduled HPC batch jobs with an ML model [31].

VIII. Conclusions

In this work we presented contributions that overcome multiple significant bottlenecks in ML collective autotuners: training point selection, non-power-of-two data points, model testing, and data collection. Combining our contributions, we developed ACCLAiM, the first practical ML autotuner for MPI collective algorithm selection that accelerates applications even when accounting for its training overhead. We applied ACCLAiM on a leadership-class supercomputer to showcase its practicality in a large-scale production setting.

IX. Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and by the U.S. National Science Foundation via award CCF-2119069.

This research used Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory, and the resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] "MPICH." [Online]. Available: https://www.mpich.org

[2] "OSU micro-benchmarks." [Online]. Available: https://mvapich.cse.ohio-state.edu/benchmarks/

[3] P. Balaprakash, M. Salim, T. D. Uram, V. Vishwanath, and S. M. Wild, "DeepHyper: Asynchronous hyperparameter search for deep neural networks," in *2018 IEEE 25th international conference on high performance computing (HiPC)*. IEEE, 2018, pp. 42–51.

[4] M. Chaarawi, J. M. Squyres, E. Gabriel, and S. Feki, "A tool for optimizing runtime parameters of Open MPI," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2008, pp. 210–217.

[5] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI usage on a production supercomputer," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 386–400.

[6] B. Efron and C. Stein, "The jackknife estimate of variance," *The Annals of Statistics*, pp. 586–596, 1981.

[7] 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 *Euro PVM/MPI*, 2006.

[8] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: self tuned adaptive routines for MPI collective operations," 01 2006, pp. 199–208.

[9] S. Gorlatch, C. Wedler, and C. Lengauer, "Optimization rules for programming with collective operations," in *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*. IEEE, 1999, pp. 492–499.

[10] 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 *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–9.

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

[12] 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)*. IEEE, 2018, pp. 45–50.

[13] ——, "Autotuning MPI collectives using performance guidelines," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2018, pp. 64–74.

[14] Intel, "Intel application performance snapshot (aps)." [Online]. Available: https://software.intel.com/sites/products/snapshots/applicationsnapshot

[15] F. Isaila, P. Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, and P. Hovland, "Collective I/O tuning using analytical and machine learning models," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 128–137.

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

[17] B. Mohammed, I. Awan, H. Ugail, and M. Younas, "Failure prediction using machine learning in a virtualised HPC system and application," *Cluster Computing*, vol. 22, no. 2, pp. 471–485, 2019.

[18] E. Nuriyev and A. Lastovetsky, "Accurate runtime selection of optimal MPI collective algorithms using analytical performance modelling," *arXiv preprint arXiv:2004.11062*, 2020.

[19] 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 *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Int'l Conference on Supercomputing (WSSPE)*, 2013.

[20] 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.

[21] S. Pellegrini, J. Wang, T. Fahringer, and H. Moritsch, "Optimizing MPI runtime parameter settings by using machine learning," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2009, pp. 196–206.

[22] J. Pjesivac-Grbovic, "Towards automatic and adaptive optimizations of MPI collective operations," 2007.

[23] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[24] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "MPI collective algorithm selection and quadtree encoding," *Parallel Computing*, vol. 33, no. 9, pp. 613–623, 2007.

[25] S. Shudler, Y. Berens, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf, "Engineering algorithms for scalability through continuous validation of performance expectations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1768–1785, 2019.

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

[27] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically tuned collective communications," in *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 2000.

[28] S. Wager, T. Hastie, and B. Efron, "Confidence intervals for random forests: The jackknife and the infinitesimal jackknife," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1625–1651, 2014.

[29] 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

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

[31] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "RLScheduler: An automated HPC batch job scheduler using reinforcement learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)*. IEEE Press, 2020.