A FACT-based Approach: Making Machine Learning Collective Autotuning Feasible on Exascale Systems

Michael Wilkins
Northwestern University
wilkins@u.northwestern.edu

Yanfei Guo
Argonne National Laboratory
yguo@anl.gov

Rajeev Thakur
Argonne National Laboratory
thakur@anl.gov

Nikos Hardavellas
Northwestern University
nikos@northwestern.edu

Peter Dinda
Northwestern University
pdinda@northwestern.edu

Min Si
Facebook
msi@fb.com

***Abstract*—According to recent performance analyses, MPI collective operations make up a quarter of the execution time on production systems. Machine learning (ML) autotuners use supervised learning to select collective algorithms, significantly improving collective performance. However, we observe two barriers preventing their adoption over the default heuristic-based autotuners. First, a user may find it difficult to compare autotuners because we lack a methodology to quantify their performance. We call this the *performance quantification challenge*. Second, to obtain the advertised performance, ML model training requires benchmark data from a vast majority of the feature space. Collecting such data regularly on large scale systems consumes far too much time and resources, and this will only get worse with exascale systems. We refer to this as the *training data collection challenge*.**

To address these challenges, we contribute (1) a performance evaluation framework to compare and improve collective autotuner designs and (2) the Feature scaling, Active learning, Converge, Tune hyperparameters (FACT) approach, a three-part methodology to minimize the training data collection time (and thus maximize practicality at larger scale) without sacrificing accuracy. In the methodology, we first preprocess feature and output values based on domain knowledge. Then, we use active learning to iteratively collect only necessary training data points. Lastly, we perform hyperparameter tuning to further improve model accuracy without any additional data. On a production scale system, our methodology produces a model of equal accuracy using 7.41x less training data collection time.

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

I. Introduction

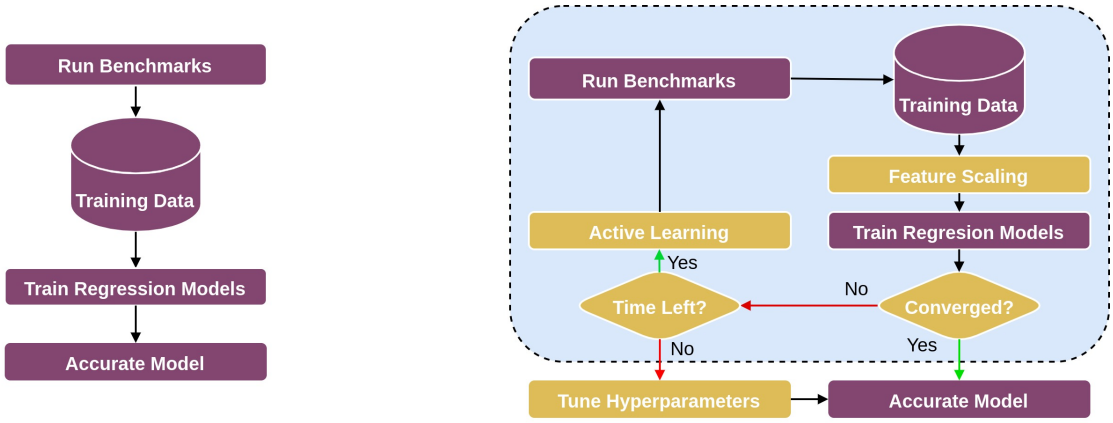
The Message Passing Interface (MPI) is the de-facto standard for communication within high-performance computing (HPC) applications. We focus this work on MPICH, a popular open-source implementation of MPI. As HPC systems and programs grow at scale, communication speed lags behind. In modern production systems, 50% of execution time is spent on MPI, not actual computation [5]. We expect communication overhead will become an even larger bottleneck in future (exascale) machines.

Collective operations are one of the most commonly used primitives in MPI, and they account for over half of MPI's overhead on production systems [5]. Collectives are a powerful abstraction because they allow programmers to specify the communication pattern of entire groups of processes in a single command. Unlike point-to-point communication, this higher-level specification conceals the actual communication pattern from the user, thus creating uncertainty in their performance. MPI implementations include multiple algorithms for the same collective. The best algorithm for a given collective is highly dependent on the situation, and using the wrong algorithm can significantly hinder performance.

Collective performance is influenced by many factors. For example, there are programmatic values such as message size, number of processes, and number of processes per node. In addition, a multitude of other factors exist (e.g., hardware performance, network design, library versions, etc.). It is exceptionally difficult for a developer to consider all seen and unseen factors and pick the right collective implementation for their use case. To address this challenge, MPICH automates collective algorithm selection using decade-old heuristics. In our studies, we found that an algorithm selected using the default heuristics was 1.3x slower than the optimal choice on average for our test system.[1]

There are many proposed methods to improve collective algorithm selection. Much work has focused on analytical models that can project algorithm performance [6], [11], [17]. These models show varying performance results, and they have failed to gain widespread adoption because they are difficult to implement, maintain, and expand for new algorithms. In production, tools such as Intel's MPITune and Open MPI's OPTO [4] use exhaustive benchmarking to automatically tune algorithm selection. This strategy maximizes accuracy, but it requires so much data (and thus machine time) that it can

[1]Bebop cluster at Argonne, 64 node subset of 664 nodes, Intel Xeon-E5-2694v4 and 128GB of DDR4 memory per node



a) Current Training Methodology

b) FACT Training Methodology

Fig. 1: System diagrams comparing our training methodology to the existing design

only be deployed to tune individual scenarios on large scale systems.

Machine learning (ML) can improve upon exhaustive approaches by learning to predict scenarios that have not been benchmarked [9], lessening the benchmarking overhead. ML also has an inherent advantage over analytical models because it can learn patterns in the data caused by factors that are difficult to model analytically, such as real-time and/or machine-specific influences. While ML has the potential to improve collective algorithm selection, the state-of-the-art ML system [7] is not ready for production use. To begin, it is difficult for a prospective user to compare ML autotuners and other existing approaches because there is no framework to quantify performance differences. We refer to this issue as the *performance quantification challenge*. Also, we observe that the state-of-the-art ML autotuner still requires an intractable amount of benchmarking data for training. We refer to this issue as the *training data collection challenge*. The evaluation in [7] uses an ad-hoc method to generate training and test sets with a small number of nodes (up to 48). The proposed method is a strong proof of concept, but it does not generalize to encompass the performance of an entire system. The standard approach would collect data for the entire feature space, then randomly split the data into training and test sets. This process seems straightforward, but we estimate collecting this data for our target system of 512 nodes takes approximately 75,000 core hours, which is over 6 days of machine time.[2]

In this paper, we address the performance quantification and training data collection challenges. To better understand autotuner performance, we define a framework of metrics that summarize how a collective autotuner's selections affect application execution time. We apply this framework to the state-of-the-art ML autotuner while decreasing the training set size and show that the performance benefit of ML disappears when using a realistic amount of data. To correct this problem, we introduce Feature scaling, Active learning, Convergence, Tune hyperparameters (FACT). FACT is a customized three-part methodology that uses machine learning techniques (e.g., hyperparameter tuning) to minimize training data collection time. First, we use expert knowledge to preprocess the feature and output values, which helps the ML model better understand the proper trends in the data. Next, we iteratively generate training data using active learning until the model converges to near-optimal selection accuracy. Finally, we apply an automated hyperparameter tuning tool that further improves model accuracy without collecting additional data. Our complete training methodology, contrasted with the current method, is shown in Figure 1. We show that, on a production scale system, our FACT-based approach produces a model of equal accuracy using 7.41x less training data collection time compared to existing approaches.

The following are the contributions of this paper:

- Metrics to evaluate collective autotuner performance;
- The FACT methodology, which includes custom preprocessing, active learning, and hyperparameter tuning to minimize the training data collection time;
- A simulated FACT-based autotuner that includes near optimal performance while utilizing a practical amount of training data.

II. Background and Challenges

We now describe the concept of MPI collective autotuning using machine learning, and its key challenges.

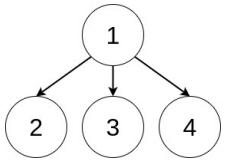
A. Collectives

Collectives work by abstracting away the point-to-point primitives typically associated with message passing. For example, consider a developer who wishes to send input data from the root process to the rest of the processes in the program. The collective primitive "broadcast" (*MPI_Bcast*) simplifies the program by performing the entire communication in a single function call. The ease of using collectives becomes increasingly important as the number of processes continues to grow. For this reason, we expect collectives will be even more popular in future exascale applications.

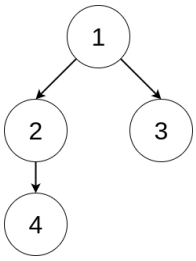
[2]For machine learning-based autotuning of MPI collective communication, the cost of data collection dominates the machine learning inferencing and prediction costs, which are therefore both negligible.

TABLE I: Types of variables that affect collective algorithm performance

Type of Variable	Examples
Programmatic	Message Size Number of Processes (N) Processes per Node (PPN)
Non-Programmatic	Library Versions Node Topology CPU Performance Network Bandwidth/Latency/Congestion



(a) Serial Implementation



(b) Binomial Tree Implementation

Fig. 2: Two possible implementations of MPI_Bcast

Collectives are beneficial because they are easy to use, but they risk deteriorating performance. The higher-level of abstraction creates ambiguity regarding the actual implementation of the requested communication. For example, MPICH includes 3–4 different algorithms for each primitive, which perform differently depending on a wide array of factors.

We separate these factors into two categories shown in Table I. First are programmatic variables that are manually set in software. The programmatic variables we consider are message size, number of processes (N), and processes per node (PPN). The other category is non-programmatic variables, which are innate to the hardware and software outside the target application. All of these factors, of both categories, must be weighed when selecting the optimal algorithm for a given collective; the wrong choice can result in a slowdown by a factor of two or greater.

To illustrate the complexity of algorithm selection, we continue our example using *MPI_Bcast*. Figure 2 shows two choices of implementations. Figure 2a is a serial broadcast: root node 1 sends the message to nodes 2, 3, and 4 in order. Figure 2b shows a binomial tree implementation where child nodes forward the message in parallel. In this case, node 2 can send the message to node 4 while node 1 sends the message to node 3. It may seem like the tree algorithm is always the superior choice. However, imagine a system with a high-latency network. Using a non-blocking send, node 1 may be able to push all of its messages into the network before node 2 even receives the information. In this case, the serial implementation could perform faster. Additionally, these are not the only algorithm choices. For large message sizes, an algorithm that performs a scatter followed by an allgather has been shown to be superior [16]. Through this discussion, we can see that this single choice alone requires an inordinate amount of expert knowledge. In addition, many applications are written to set programmatic values dynamically, making it impossible for the developer to make the correct selection. To address the challenge of collective algorithm selection, autotuning is a natural solution.

B. Collective Autotuning with Machine Learning

The challenge of collective algorithm selection has inspired many autotuner proposals, which we describe in more detail in Section VI. Here, we discuss existing machine learning approaches and their challenges. As a prerequisite, we adopt the autotuner described in [7]. We create a machine learning regression model for every algorithm of every collective. We experimented with the most performant machine learning models from previous works (Random Forest, XGBoost, GAM, KNN) and found that Random Forest worked best with our data on our system. This choice is arbitrary, as others such as XGBoost and K-Nearest Neighbors produce similar results. The learners accept the 3 programmatic values (message size, N, PPN) as input features. The output of the model is a predicted execution time in microseconds. The goal of each learner is to predict the execution time of its algorithm across the entire feature space. To select an algorithm for a particular feature set, we query the regression models for each algorithm and select the one with the lowest predicted execution time. This design has been shown to make accurate selections that accelerate collectives. However, there are obstacles preventing its use on production systems.

C. Unresolved Challenges

Performance quantification challenge: The researchers in [7] evaluate their work by benchmarking every algorithm for their test data points. They use this data to simulate the performance of selections by the MPI library's default method, their ML autotuner, and an oracle that always selects the best algorithm. They show sometimes significant (1.3-1.5x) speedup over the default selections. However, they do not comprehensively compare to the oracle. The omission of quantitative data makes comparisons with other autotuners more difficult. In addition, average speedup does not paint the whole picture with respect to applications. For example, programs use only a small fraction of the feature space. Average speedup could cover the weaknesses of a high-variation autotuner, which makes some very good selections and some very bad selections. If an unlucky application only uses scenarios with bad selections, they will not gain the promised speedup. This evaluation works well as a proof of concept, but there are flaws when considering production applications. We provide an evaluation framework in Section III that includes new metrics to better measure autotuner performance and its effect on applications.

TABLE II: Metrics to quantify collective algorithm autotuner performance

Metric	Definition	Range of Values
$R^2\,Score$	General statistic describing how well the model fits the data	[0,1] (closer to 1 is better)
Average Selected Algorithm Slowdown (*Average Slowdown*)	Slowdown of the selected algorithm compared to the optimal/oracle algorithm averaged across all feature sets	[1,∞] (closer to 1 is better)
Classification Accuracy	Proportion of feature sets where the ML model accurately predicts the fastest algorithm	[0,1] (closer to 1 is better)
Significant Mistake Proportion	Proportion of feature sets where the predicted algorithm is more than 10% slower than the fastest algorithm	[0,1] (closer to 0 is better)

Training data collection challenge: Previous work used an ad-hoc method to create training and testing datasets on systems of up to 48 nodes. Obviously, modern and future HPC hardware have many more nodes, and attempting to replicate the custom data collection methods from the previous work would be completely intractable. In this paper, we use a 512-node system and limit our tests to power-of-2 feature values. The common strategy used in ML is to measure the entire feature space and then randomly split the data for training and testing. For our large scale experiments, we collected this data for one of the most popular collectives (*MPI_Bcast*), and we estimate that doing so for all 12 standard (non-variable, blocking) collective operations would take 75,000 core hours (over 6 days of machine time). For exascale machines, this number would be significantly larger. In a production system, training would have to be repeated regularly to account for changes in non-programmatic variables, multiplying the true cost of training data collection. We present the FACT methodology in Section IV that minimizes the training data collection time, making it more feasible on exascale machines.

III. Quantifying Performance

To address the performance quantification problem, we propose the set of metrics in Table II. In this section, we explain the importance of each metric. We then use them to re-analyze the performance of the existing work and illustrate the training data collection problem.

A. Metrics

We believe each metric encapsulates an important dimension of autotuner performance. First is $R^2\,Score$, a generic value that represents ML model fitness. An $R^2\,Score$ close to one means the individual regression models capture the trends in the dataset. This metric is useful because it indicates how the model may perform on untested scenarios or more difficult areas of the feature space.

The most commonly used metric is *Average Selected Algorithm Slowdown*, or *Average Slowdown* for short. *Average Slowdown* represents the expected inefficiency of the autotuner's selections compared to optimal. This metric is most useful when making a comparison between autotuners because it represents the performance of selected collective algorithms across the entire feature space.

The next metric is *Classification Accuracy*, which represents the chance that a selection will be the optimal algorithm. This metric is useful to measure model performance in the presence of outliers. For example, a model may perform very poorly for edge cases that go unused in applications. This model would have a poor *Average Slowdown*, but the *Classification Accuracy* would more fairly represent its high performance.

Lastly, we include *Significant Mistake Proportion* to represent a chance that a selected algorithm will be significantly ($> 10\%$) slower than the optimal selection. *Significant Mistake Proportion* is the best statistic to decide whether an autotuner is "good enough" for users because it shows the chance a selected algorithm will perform noticeably worse than it should. Because applications only use a small fraction of the feature space, a small *Significant Mistake Proportion* guarantees that they will see near-optimal performance from the autotuner. It is important to note that 10% is an arbitrary value and can easily be changed.

The metrics included in Table II paint a fuller picture of autotuner performance. We feel it is important to address the most obvious omission: *Average Speedup*. *Average Speedup* is the predominant metric used in existing works because it shows how much better an autotuner's selections perform compared to a library's default selection mechanism. We disregard it because, at this point, the poor performance of default selections is a well-known fact and beating them is no longer interesting. Moreover, default selection is much better on some hardware systems than others. Attempting to compare autotuners from separate sources based on *Average Speedup* is pointless because it depends more on the difference in default performance. Instead, our metrics should be applied separately to an autotuner and the default and then compared.

B. Previous Work Evaluation

To illustrate the usefulness of our metric set, we re-implemented the existing work in [7]. We exhaustively benchmarked the standard collectives for all power of two feature values up to 64 nodes, 32 process per node, and 1 MB messages. We selected 64 nodes because it is the closest power of two greater than the 48 nodes used for evaluation in [7]. We trained the ML autotuner using randomly selected training data. The test set includes all of the data points we collected. We used the *RandomForestRegressor* from the *scikit-learn* package [12] as our ML model. Because we benchmarked every algorithm in MPICH for every feature set, we know which algorithm is optimal in every scenario. We then calculated our metrics using the autotuner's selections and the optimal selections. For comparison, we also simulated the selections MPICH would make by default and found an *Average Slowdown* of 1.3.[3]

We performed these experiments on the Bebop cluster at Argonne National Laboratory. We used a 64-node subset of

[3]Note that this means that there is headroom to improve on the MPICH selections and speed up collectives by 23.1%.

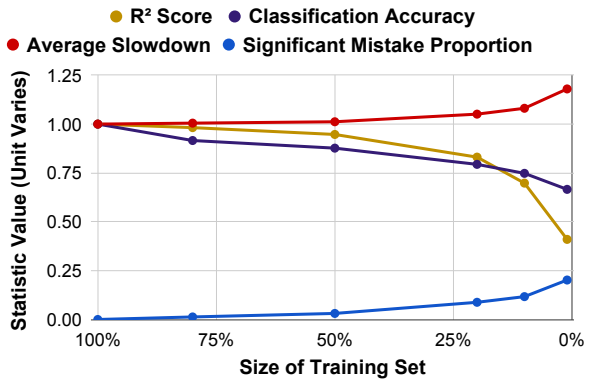


Fig. 3: State-of-the-art performance as a function of training set constriction.

the 664 standard nodes. Each node contains an Intel Xeon-E5-2694v4 with 36 cores (we use up to 32 cores) and 128GB of DDR4 memory. The results of our evaluation are summarized in Figure 3. We repeated the experiment with smaller training sets to understand how the ML autotuner performs when we train it with more realistic amounts of data. The left side of the graph confirms the results from the previous work: the ML autotuner performs exceptionally well with copious training data. With an *Average Slowdown* near 1 and a *Significant Mistake Proportion* near 0, applications would enjoy near optimal collective performance. However, the picture becomes increasingly tainted as we move to the right in the table. The results stay stable for 50%, steadily deteriorate at 20/10%, and completely collapse at 1%. 10% is the upper limit of what may be feasible on a larger scale system (1.82 machine hours, see Figure 9). With 10% of the feature space for training, an *Average Slowdown* of 1.10 is substantially suboptimal, and a 0.12 *Significant Mistake Proportion* means many applications would see noticeable performance deterioration.

We conclude that when trained with a realistically sized training set, the state of the art provides far less practical value. These results showcase the training data collection problem.

IV. Minimizing the Training Set

To address the training data collection problem, we propose the FACT methodology. Here, we describe the three components of FACT. We conclude the section by integrating each segment to create the training methodology in Figure 1.

A. Feature Scaling

Data preprocessing is a ubiquitous step in machine learning applications. Preprocessing allows the developer to use domain knowledge to help expose data patterns to the ML models, greatly improving model accuracy. More specifically, the developer processes the data to eliminate anomalies that may mislead the learner. One of the most common preprocessing steps is feature scaling. Feature scaling is vital for collective autotuners because some regression models typically treat larger feature values as more important by construction. In our case, message size has a much bigger range than number of nodes or processes per node, but we want the model to treat them all equally.

TABLE III: Feature ranges before and after scaling

	Range	Scaled Range
N	[1,512]	[1,10]
PPN	[1,32]	[1,6]
MSG SIZE	[1, 1MB]	[1,21]

To reduce our feature ranges equitably, we take advantage of our assumption that all features values must be powers of two. Therefore, a log_2 scaler is the obvious solution. We then add one to avoid feature values of zero, which are also known to confuse some models. Through our feature scaling approach, their ranges become much more similar. We summarize the ranges before and after scaling in Table III.

Scaling the input values has a natural solution, but the output values present a more complicated case. To begin, we observe that the range of output values is quite large, from a few microseconds for small inputs to a full second or more for large inputs. In this scenario, a regression model may treat outputs for small feature values as essentially the same. However, our metrics normalize the output values for each feature set to the optimal algorithm. A difference of a couple of microseconds may be a significant slowdown/speedup for small feature values, and we need to make sure the model maintains that information.

The most common approaches for scaling are standardization and normalization. In short, standardization assumes the data fits a normal distribution and rescales it to a mean of 0 and a standard deviation of 1. Normalization is a uniform scaling technique that compresses the data into the range [0,1] without affecting the shape of its distribution. Another approach to consider is copying the metrics and scaling each output to the value of the fastest algorithm. We refer to this technique as *algorithm scaling*. Finally, we could add a log_{10} scaler to algorithm scaling to match the scaling pattern already applied to the inputs.

To evaluate our options, we repeat the experiments from Figure 3 with input scaling and each of the output scaling options applied. The results are shown in Figure 4. To summarize, our custom solution applying log_{10} with algorithm scaling outperforms the competitors. To understand why, we discuss the pitfalls of each option. Standardization performs poorly because it assumes that the data fits a normal distribution, which is not true for our data set. Normalization fails because our input scaling complicates the input-output relationship by artificially introducing an exponential component, and normalization passes this complexity on to the ML model. Given the weaknesses of the other preprocessing schemes, our custom solution using algorithm scaling with an additional log_{10} scaler is the clear winner. We believe it performs well because it scales the output to make relative differences in performance more important, and it also eliminates the exponential relation introduced by our input scaling technique.

By combining our custom feature scaling techniques, we observe a significant improvement in model performance. Figure 5 compares our preprocessing to the original solution,

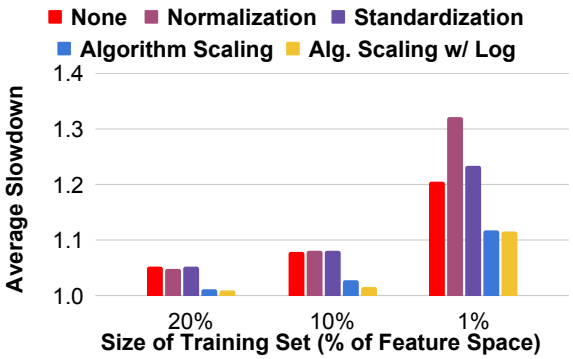


Fig. 4: Effects of different preprocessing techniques on *Average Slowdown* of resulting selection as a function of training set constriction. Alg. Scaling w/ Log performs the best for every training set size.

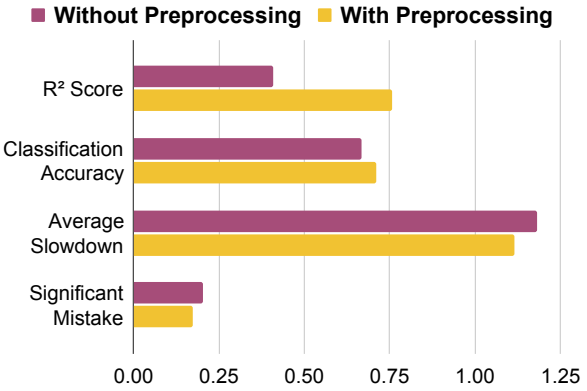


Fig. 5: Effect of using our custom preprocessing technique. The resulting model sees significant improvement across all metrics.

which did not preprocess the data. We observe improvements across all metrics in Figure 5. With a feasible amount of training data (1% of the feature set), *Average Slowdown* decreases by 35.7%. Meanwhile, *Significant Mistake Proportion* decreases by 14%. Interestingly, *Classification Accuracy* only sees a relatively modest uplift (4.3%). By considering the trends across statistics, we can conclude that our preprocessing improves the model not by making more correct selections, but by making smaller mistakes when it does miss-select.

B. Active Learning

Active learning is a type of machine learning where the learner interactively queries the data set [15]. It is an iterative process that begins with a set of unlabeled data. In each iteration, the learner chooses data points to be labeled by an oracle and adds them to its training set. Then the process repeats, and the model selects new points. Training continues until the model accuracy converges or a time limit is reached. Active learning is typically used for ML applications where data labeling is a challenging or time consuming task. Examples typically include situations that require human intervention (e.g., text/speech/image recognition), but the same description applies to the training data collection problem.

For point selection, we use the most common strategy: uncertainty sampling [10]. This algorithm queries data points that it is most uncertain how to label. We use a surrogate model [**?**] to represent the distribution we are attempting to learn. The purpose of the surrogate model is to fit the distribution and report features values at which it is most uncertain.

To use active learning to train a collective autotuner, we specify an unlabeled data space using our input features and their scaled ranges. During each iteration, we look up the execution times of the chosen points using our exhaustive benchmark results to simulate data collection. After each iteration, we test model accuracy to check if we have met the convergence criteria. This threshold is set by the user according to their preferences. Example criteria include *Average Slowdown* below 1.03 or *Significant Mistake Proportion* below .05.

C. Hyperparameter Tuning

Hyperparameter tuning is a process where the parameters of the learning model (hyperparameters) are optimized. All of the most common learners have many hyperparameters, and using the optimal values can greatly improve model accuracy. For example, hyperparameters of the random forest model used in this work include the number of decision trees and the max depth of the trees.

In theory, hyperparameter tuning should be performed every time we train our regression models, so as to maximize the accuracy we can gain from the data collected. However, similar to the data set collection problem we are attempting to solve, searching the hyperparameter space is time consuming. To perform hyperparameter tuning during our active learning iterations before testing for convergence, we would have to pause data collection while we retrain the model with many combinations of hyperparameter values. In our experience, we found that collecting more data improved model accuracy far quicker than hyperparameter tuning. For this reason, we include hyperparameter tuning as a postprocessing step in our methodology. This way, it can be performed offline or on a single node, not expending the data collection resources. During active learning, we use fixed hyperparameter values that are derived from previous offline tunings. Hyperparameter tuning is then an optional step to squeeze the last bit of accuracy out of the model after data collection. This step is most valuable when the user only has a fixed amount of time to collect training data. In this case, the active learning process may not converge, resulting in inaccurate models. Hyperparameter tuning can bridge the gap and produce a model with converged-level accuracy. It is through this lens we evaluate hyperparameter tuning in Section V.

D. Implementation

Our FACT implementation integrates the main three ideas from this section.

We support data collection using both the Ohio State University (OSU) microbenchmark suite [1] and the ReproMPI

benchmark suite [8]. The OSU benchmarks are a widely accepted suite for benchmarking MPI collectives, and they are the benchmarks we use in our evaluation. Once the data is collected, we preprocess the data and use it to train the *RandomForestRegressor* included in the *scikit-learn* Python package [12].

For active learning, we deploy a special instance of the DeepHyper tool [2], [3]. DeepHyper is primarily an automatic hyperparameter tuning tool. It works by iterating through hyperparameter configurations, training an underlying (surrogate) model. The surrogate model maps the hyperparameter configurations to a result statistic defined by the user (e.g., classification accuracy or R^2 score). DeepHyper balances two modes: *exploration* and *exploitation*. During the exploration phase, it queries the surrogate for the hyperparameter values with the most uncertainty. It then trains the target model with those values, tests its performance based on the user-defined metric, and uses the results to retrain the surrogate model. The "exploitation" phase then uses the surrogate model to predict which hyperparameter values will maximize the performance of the target model. Figure 6 illustrates DeepHyper's inner workings.

We observe that the exploration phase of DeepHyper is very similar to the active learning process. We map our problem onto DeepHyper's framework as follows:

- We set DeepHyper's β value to ∞, which forces DeepHyper to always stay in exploration mode.
- We define the performance criteria as the execution time of the collective algorithm.
- We specify our input features (N, PPN, message size) as the hyperparameters for DeepHyper to optimize.
- To train the target model, we instruct DeepHyper to run the selected benchmark suite and report the execution time.

By manipulating DeepHyper, we implement our active learning methods with much less development effort than an ad-hoc approach.

For hyperparameter tuning, we apply DeepHyper for its intended purpose. We tune the following hyperparameters for the random forest model: number of trees, split criterion, max tree depth, and minimum number of samples to split a node.

Currently, our data collection and machine learning techniques are separate. For now, simulate the iterative process of collecting data and retraining by looking up the benchmark results from our previously collected exhaustive results.

V. Evaluation

In this section, we compare our implementation's performance with the performance of the existing work. Then, we use a much larger evaluation platform to show the benefits of the FACT methodology at scale. By reducing the data collection time, especially on larger scale machines, FACT makes ML-based collective algorithm autotuner more feasible on exascale systems.

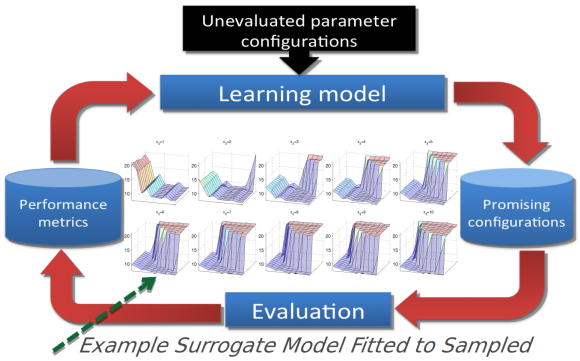


Fig. 6: DeepHyper System Diagram [3]

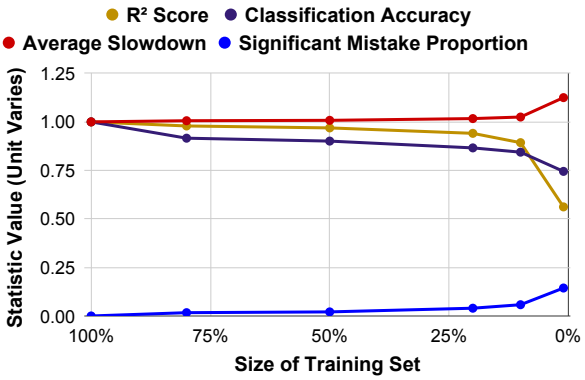


Fig. 7: FACT-based implementation performance. A FACT-based approach greatly improves autotuner performance with smaller training sets.

A. Existing Work Comparison

For comparison, we recreated Figure 3 using our FACT implementation. Again, we used a 64 node subset of the Argonne Bebop cluster, each containing an Intel Xeon-E5-2694v4 with 36 cores (we use up to 32 cores) and 128GB of DDR4 memory. The result is shown in Figure 7.

Given the exact same exhaustive dataset, the FACT-based approach greatly improves performance with smaller training sets. For training sets between 50% and 1% of the feature space, *Average Slowdown* decreased by 33-68% compared the previous state of the art. *Classification Accuracy* (3-13%) and *Significant Mistake Proportion* (32-55%) also saw substantial improvements.

By improving autotuner performance with small training sets, FACT minimizes data collection time. As a prerequisite, we apply our previous example convergence criteria. We consider the model "converged" if *Average Slowdown* is below 1.03 and *Significant Mistake Proportion* is below .05 The FACT-based approach reaches the criteria with a training set of roughly 10% of the feature space, while the existing work requires roughly 50% of the feature space. It takes 25.8 machine hours to collect the 50%, randomly selected training set. The 10%, active learning training set takes 3.76 machine

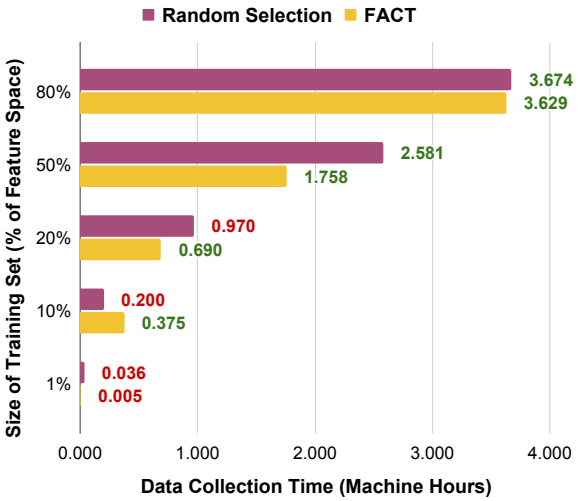


Fig. 8: 64 Node Data Collection Time for All Collectives. Training sets with green labels result in ML models that meet the convergence criteria, while those with red labels do not. On average, FACT reduced data collection time by an average of 1.14x for training sets of the same size.

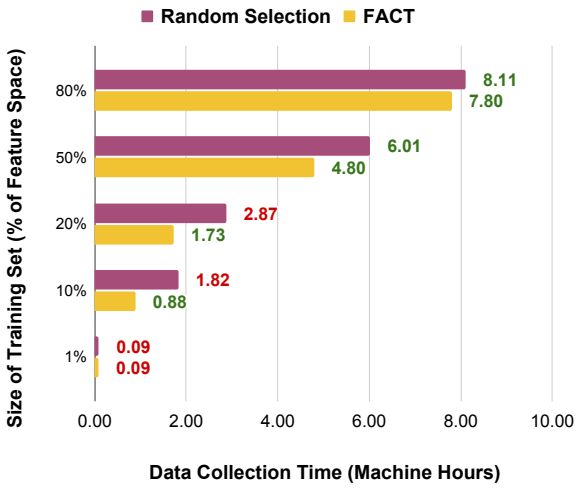


Fig. 9: 512 Node Data Collection Time for *MPI_Bcast*. Training sets with green labels result in ML models that meet the convergence criteria, while those with red labels do not. On average, FACT reduced data collection time by an average of 1.31x for training sets of the same size.

hours to collect. The 7.41x reduction is more than expected based on the set size decrease (5x).

Active learning improves data collection time even more than the training set size suggests because it is naturally biased towards smaller feature values, which take less time to collect. Feature sets with smaller values have more variation. It follows that the surrogate model assigns greater uncertainty values in this range, therefore choosing these points instead of larger feature values. This effect is even greater with the smallest training sets. When both methods use 1% of the feature space, the active learner collects its data in 7.2x less time (.05 hours compared to .36 hours). Full results for data collection time are shown in Figure 8.

B. Towards Exascale Systems

To understand how our training time minimization techniques scale with machine size, we also apply our implementation to a larger scale test system. We used a 512-node subset of the 4,392 node Theta supercomputer at Argonne. Each node is comprised of an Intel Xeon Phi 7230 with 64 cores (we again use up to 32 cores) and 192 GB of DDR4 memory. At larger scale, it is impractical to collect exhaustive data for all collectives as we did in Section III. We instead collect data for one of the most popular collectives: *MPI_Bcast* [5].

Using our large scale *MPI_Bcast* data, we plotted the data collection time for each training set size in Figure 9. Note that this figure only shows the data collection time for *MPI_Bcast*, while Figure 8 shows the data collection time for all standard, non-blocking collectives. We use *MPI_Bcast*'s increase in data collection time from 64 to 512 nodes times cumulative results up to 64 nodes on Theta to calculate the machine/core hour estimates in the introduction.

On the 512 node production scale machine, the benefits of FACT are amplified. Assuming the convergence criteria are met with the same amounts of data as the previous experiment[4], FACT requires 6.8x less training time (6.01 hours to .88 hours). Even on larger systems, FACT continues to greatly decrease data collection time.

C. Hyperparameter Tuning

To understand the benefits of hyperparameter tuning, again consider the scenario where the user does not have enough machine time to generate a converged model. For this experiment, we state our convergence criteria to be when the model has an average slowdown less than 1.02.

We represent the convergence in Figure 10, which shows how *Average Slowdown* decreases as the active learner adds points to the training set. This data is generated from *MPI_Reduce_Scatter* from our 64 node testcase. We chose *MPI_Reduce_Scatter* because it is one of the slowest collective operations to converge, making it one of the most likely to require hyperparameter tuning. The first line indicates where the user ran out of data collection time (22.2 machine minutes), and the second line represents the model convergence point if they had been able to continue training (36 machine minutes).

We run DeepHyper's hyperparameter tuning routine to automatically generate a better random forest configuration for our training set that minimizes average slowdown. The results of the most important iterations are shown in Table IV. We see that the tuning process generates a set of hyperparameters that produce an ML model that reduces the average slowdown by 2x, getting under the convergence target in less than two-thirds the data collection time.

[4]For large scale job allocations, the amount of data required to meet the convergence criteria varies widely, both across different jobs and over time within the same job. For more details on the difficulties of microbenchmarking MPI, see Hunold et al [8]. The best way to fully measure performance on large scale jobs is through applications, which we leave as future work.

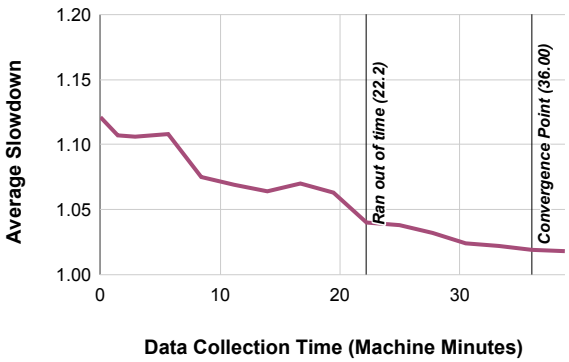


Fig. 10: Example Convergence Graph. The user runs out of data collection time when they are only halfway to convergence.

TABLE IV: Hyperparameter Tuning Iterations

Iteration	*Average Slowdown*
1	1.04
2	1.057
...	...
20	1.019

Note that for this experiment, DeepHyper ran for around 15 minutes to find performant hyperparameter values. Attempting to introduce this process between every data point we collect would increase the overall time by 15 minutes times the number of training points collected. In the scenario with limited collection time, attempting to run hyperparameter tuning during active learning would inhibit so much data collection that we would never get enough data to create a converged model, regardless of the hyperparameter values.

VI. Related Work

This paper builds upon the ideas presented by Hunold et al. [7], [9], who present the idea of using machine learning to autotune collective algorithms. They prove that basic machine learning models without hyperparameter tuning can accurately select collective algorithms. The work in [7] goes further, building an autotuner prototype using basic ML models without hyperparameter tuning. Their paper includes an ad hoc testing methodology that shows an ML autotuner works well on allocations up to 48 nodes on larger supercomputers. We scale to much larger node counts while minimizing the data collection time using the FACT methodology.

There are many previous proposals that use non-ML methods to autotune collective algorithms. Some work focuses on modeling collective algorithm performance using analytical models [13]. Early autotuners use heuristics and/or analytic models to select collective algorithms and tune their parameters [4], [6], [14], [17]. A more recent study by Luo et al. used a hierarchical model to optimize algorithms [11]. They create tasks (e.g., a portion of a collective algorithm) using swappable submodules. The submodules exist at lower-level hardware layers, making it easier to support heterogeneous hardware. Old and new analytical autotuners suffer from similar issues: high implementation, maintenance, and expansion costs that prevent them from appearing in production-ready MPI implementations. Machine learning, on the other hand, handles the complexity of algorithm selection internally. The black box nature of machine learners make them much easier to use and expand to fit new algorithms. Our work and previous ML studies have confirmed that ML models are capable of accurately predicting the performance of collective algorithms without exposing the user or developer to undue burden.

VII. Future Work

The work presented in this paper addresses the most pressing challenges preventing ML autotuners from being adopted in production. However, there are still existing challenges that we discuss here.

A. Effect on Application Performance

Collective performance is important to applications because it constitutes a large portion of the total execution time [5]. However, it is unclear exactly how much accelerating collectives would accelerate an entire application. An example complication is communication-compute overlap. If each node can make progress while waiting for a collective operation to finish, improving the collective latency may not affect the overall application speed in a meaningful way.

In Section III, we propose a set of metrics that better capture collective performance and its potential effects on applications. To fully understand the correlation between our metrics and application performance, we need to test autotuner collective algorithm selections in the context of real applications on large scale systems. We require a full autotuner prototype to perform these experiments.

B. Building a Complete Autotuner

Our contributions make an ML autotuner more feasible on a production system. However, we do not actually build a complete autotuning system in this paper. Our current prototype simulates the active learning process (see Section IV-D). As a next step, we plan to build a complete prototype that implements our ideas. For this prototype to be useful, we must address additional challenges introduced by our design.

Non-power-of-two: A major assumption we make in this paper is that all feature values (number of nodes, etc.) are powers of two. In practice, these values do not always meet this assumption. Number of nodes is the most common deviant. Many large scale machines (e.g., our test platform) do not have a power-of-two number of nodes. Also, these systems prioritize utilization in their job scheduling algorithms. For these reasons, non-power-of-two jobs are commonly promoted by the schedulers because they help fill every node.

To address non-power-of-two feature values, we plan to include non-power-of-two situations in the active learning process. By occasionally sampling non-power-of-two feature values with high uncertainty, we can accurately predict both power-of-two and non-power-of-two feature values without substantially increasing data collection time.

Convergence: In this paper, we use metrics measured across the exhaustive dataset to detect when a model has converged. In practice, we will not have a complete set of data to compare against. Instead, we will only have the points collected for training, which we cannot reuse for testing.

We need a new way to detect convergence in an active learning autotuner. We plan to find a correlation between the convergence metrics we currently use and the uncertainty values of our active learner (see Section IV-D). DeepHyper's implementation does not expose these values to the user. In addition, DeepHyper runs for a fixed number of iterations, not until a criteria is met. To overcome these challenges, we will develop our own custom active learning tool. With a custom implementation, we can continue active learning until a level of certainty is met. We can verify the correlation by doing active learning first on a new system then collecting exhaustive test data to double-check accuracy.

After solving the non-power-of-two and convergence challenges, we will build a production-ready ML autotuner that minimizes data collection time.

C. Practicality over Default Approaches

With a complete autotuner in place, a 7.41x reduction to the estimated 6 days of data collection time is still almost a day of machine time. Considering training data must be recollected as frequently as every job allocation, FACT-based collective autotuning is only practical for longer-running jobs. We plan to continue addressing the issue through strategies such as parallelized data collection. In the context of real applications, we plan to analyze whether the improved performance makes up for the data collection overhead.

VIII. Conclusions

We presented an evaluation framework for MPI collective autotuners. We use this framework to showcase the FACT approach. FACT can generate an ML-based autotuner with performance equal to the state of the art. FACT maintains performance while minimizing the training data collection time, making ML-based autotuners more feasible on exascale supercomputers. Moving forward, we believe a complete FACT-based autotuner will maximize collective performance with minimal data collection, while also minimizing maintenance and expansion costs.

IX. Acknowledgements

We would like to acknowledge the team behind DeepHyper, specifically Prasanna Balaprakash and Jaehoon Koo, for their assistance configuring DeepHyper for our experiments.

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, and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

We acknowledge the computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

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

References

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

[2] P. Balaprakash, R. Egele, M. Salim, S. Wild, V. Vishwanath, F. Xia, T. Brettin, and R. Stevens, "Scalable reinforcement-learning-based neural architecture search for cancer deep learning research," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–33.

[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] 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.

[7] 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.

[8] S. Hunold and A. Carpen-Amarie, "Reproducible mpi benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3617–3630, 2016.

[9] ——, "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.

[10] D. D. Lewis and W. A. Gale, "A sequential algorithm for training text classifiers," in *SIGIR94*. Springer, 1994, pp. 3–12.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] B. Settles, "Active learning literature survey," 2009, Technical Report, University of Wisconsin-Madison, Department of Computer Sciences.

[16] 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.

[17] 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.