Benchmarking Machine Learning Methods for Performance Modeling of Scientific Applications

Preeti Malakar, Prasanna Balaprakash, Venkatram Vishwanath, Vitali Morozov, and Kalyan Kumaran
Argonne National Laboratory
Email: {pmalakar, pbalapra, venkat, morozov, kumaran}@anl.gov

***Abstract*—Performance modeling is an important and active area of research in high-performance computing (HPC). It helps in better job scheduling and also improves overall performance of coupled applications. Sufficiently rich analytical models are challenging to develop, however, because of interactions between different node components, network topologies, job interference, and application complexity. When analytical performance models become restrictive because of application dynamics and/or multicomponent interactions, machine-learning-based performance models can be helpful. While machine learning (ML) methods do not require underlying system or application knowledge, they are efficient in learning the unknown interactions of the application and system parameters empirically using application runs. We present a benchmark study in which we evaluate eleven machine learning methods for modeling the performance of four representative scientific applications that are irregular and with skewed domain configurations on four leadership-class HPC platforms. We assess the impact of feature engineering, size of training set, modern hardware platforms, transfer learning, extrapolation on the prediction accuracy, and training and inference times. We find that bagging, boosting, and deep neural network ML methods are promising approaches with median R^2 values greater than 0.95 and these methods do not require feature engineering. We demonstrate that cross-platform performance prediction can be improved significantly using transfer learning with deep neural networks.**

***Index Terms*—performance modeling; machine learning; benchmarking; transfer learning**

I. Introduction

Performance models are used to obtain runtime estimates by modeling various facets (computation, communication, etc.) of an application on a target system. Developing performance models is an active area of research for application developers and system architects. Performance models are useful in many scenarios. For example, they can help performance engineers and application scientists avoid running a large number of input configurations for their codes on the target machine by predicting performance metrics of an unevaluated configuration. The input configuration may consist of application parameters, such as problem size (which can be multidimensional), and system parameters, such as node count, ranks per node, and process decompositions.

Accurate runtime estimates may result in less queuing delay for users and better resource scheduling at high-performance computing (HPC) sites. Performance modeling is also useful for coupled applications such as multiscale modeling [1], [2], and in situ analysis and visualization [3], [4]. In these cases, multiple codes execute simultaneously, often in space-shared mode as part of the same parallel job. These codes may have different scalability characteristics. It is useful to assess the appropriate number of nodes for concurrently executing these codes on subsets of the processor space. Performance models are helpful in such cases in order to quickly search and evaluate the experimental configurations.

The performance of a parallel application on a large-scale machine depends on (at the minimum) the algorithm, processor architecture, and network topology [5]. Accurate performance prediction is complex because of the unknown interaction of the application parameters and the system parameters of the parallel systems. Researchers use three major approaches for performance modeling: (1) analytical modeling of the application using precise knowledge of the code and the target system, (2) empirical modeling for an unknown set of application and system parameters based on empirical results for a known collection of application and system parameters (sample set), and (3) trace-based simulations. The best and yet the most difficult approach for predicting performance is to derive an analytical model of the application and associate it with the appropriate system model. However, performance of an application is affected by a multitude of system parameters, such as the number of arithmetic units, cache hierarchy, memory bandwidth, message injection bandwidth, network topology, process-to-processor mapping, network diameter, network bandwidth, and network routing. All these parameters lead to complex scalability behavior of parallel codes on a given system, especially on HPC platforms where several jobs are co-scheduled. Detailed analytical modeling of a large complex application is also difficult becasue of multisite multiparty collaborative software development.

An effective alternative to analytical modeling is empirical modeling. In this approach, a user-defined set of parameter configurations is evaluated on the target machine to measure the required performance metrics (execution time in our case). In this work, we predict the performance of an application by developing a model from the empirical data (samples) collected from a few application runs. Using the empirical performance model, we predict the output of a new configuration on the target machine. A promising approach for empirical modeling is machine learning (ML), in particular supervised learning.

Application of ML to performance prediction is not new, and many researchers have adapted ML approaches for various empirical performance modeling tasks. However, no single

method performs well on all types of HPC modeling tasks. This difficulty is exacerbated by the fact that there is no experimental study of a broad range of ML methods for various scientific application-performance modeling tasks. Consequently, the deployment of ML methods in HPC has become an adhoc process without experimental evidence and/or theoretical justification. In this paper, we focus on modeling the performance of a representative set of scientific applications on leadership-class systems, and we provide an unbiased experimental study of a broad range of supervised ML methods.

To assess our methodology, we experimented with representative miniapps such as miniMD, miniAMR, miniFE from the Mantevo suite [6], and the classical molecular dynamics simulation LAMMPS [7]. Miniapplications represent core algorithms implemented in many large-scale simulation codes and exhibit performance similar to that of the full applications. Authors in [8] verify the fidelity of miniapps, including miniMD and miniFE. We have not only used different input grid sizes for all the applications but also experimented with skewed 3D domains. Disproportionate 3D domains or mesh size of a problem can result in unfavorable default mapping of the application on the system, which affects performance [2]. Our model is able to predict performance of such problem configurations reasonably well. We also used irregular applications such as miniAMR that exhibit poor scalability with increased core count. We demonstrate our results on four large-scale systems: Mira and Vesta at the Argonne Leadership Computing Facility [9], and Edison and Hopper at the National Energy Research Scientific Computing Center [10], [11].

We summarize our main contributions below.

1) We benchmark a broad range of supervised ML methods for modeling the performance of various representative scientific applications (irregular and with skewed domain configurations) on leadership-class systems. We assess the impact of feature engineering and size of the training set on the prediction accuracy, impact of modern hardware platforms on model training and inference, and limitations of ML methods when faced with extrapolation and limited training data regime.
2) We develop a high-performing deep neural network architecture for the empirical performance-modeling tasks.
3) We demonstrate that cross-platform performance prediction is feasible and can be improved significantly by using transfer learning with deep neural networks.
4) We show that bagging, boosting, and deep neural network ML methods are promising approaches for empirical performance modeling.
5) We evaluate the efficacy of the supervised ML methods for predicting performance on up to 0.1 million cores and for large problem sizes in the considered applications (millions of atoms).

II. Related Work

In this section, we review some of the existing efforts to model application performance. Pllana et al. [12] surveyed various approaches for performance modeling, including measurement techniques, mathematical modeling, and detailed simulations. Allan et al. [13] compared tools for predicting application performance on a range of architectures. Many performance models use detailed analytical models of applications [14]–[16]. Kerbyson et al. [14] developed an analytical performance model for Weather Research and Forecast (WRF) model parameterized in terms of application inputs and system parameters by manually scrutinizing the dynamic execution behavior of the model. Obtaining application-specific details for a large application not only is cumbersome but also may be inaccurate because of the involvement of fairly big multisite multiparty collaborative software development groups. We propose applications models that do not require application code inspection.

Several performance models have been developed by using a combination of hardware and network models [5], [17]–[20]. Application performance may be bottlenecked, however, by interaction with the memory hierarchy or by communication performance. Wu et al. [17] modeled performance using memory bandwidth contention time and a communication model. They used MPI benchmarks to create a communication database and used that to derive communication performance. Snavely et al. [5] proposed that application performance is affected by single-processor performance and network use. In modern interconnects, network usage is often not same at all scales. Communication times may increase at larger node counts because of network congestion. They used a latency/bandwidth model for predicting network performance. Even a simple latency/bandwidth model, however, results in large prediction errors [21], especially on complex machines such as the Blue Gene/Q.

Tracing is often used to deduce system-specific performance details [21]–[24]. Carrington et al. [23] extrapolated application performance from traces collected on small core counts. They used application signature, machine profile, and a convolution method to predict performance. Collections of machine profiles require architecture-specific detailed knowledge such as the number of integer units, cache hierarchy, cache latencies, memory latency, and prefetch policy. Gather proprietary information for some processors is often difficult. They instrument the application for trace collection, which can be a few terrabytes for large numbers of processes [25], [26]. They tried to fit a linear, exponential, logarithmic model for different operations such as the number of L2 hits and number of memory operations. Our approaches do not incur time overhead of code instrumentation and tracing and space overhead of collecting traces.

The main drawback of the current approaches is the complexity in modeling the system and possibly erroneous parameters. To develop an accurate communication model, one needs to know communication parameters such as message size, process placement, routing algorithm, MPI implementation being used and network latency and bandwidth. Often not cannot accurately determine some of these parameters, especially in large systems such as the Blue Gene/Q and K-computer. More

application and system knowledge helps in better modeling, but this can become cumbersome. Our approaches predict performance reasonably well without considering complex computation and communication interactions. We use empirical measurements from a small number of time steps.

Wong et al. [24] created application signatures that can be executed on different systems, with less time overhead for empirical data collection. They extracted valuable information about the performance characteristics of an application. This can be subjective, however, the exhaustive list of parameters affecting performance is hard to decipher. They achieved high prediction accuracy, but significant effort was spent on code instrumentation, phase identification, and execution of phases. They reported overhead of 2x for some cases. It may be easier to run the application for fewer iterations for empirical data collection, as in our case. Our approach can leverage their method to decrease our experimental data collection time by using application signatures if they are publicly available.

Application-agnostic approaches such as linear regression have been used in multiple domains. Delgado et al. [27] described a regression-based approach for modeling WRF performance on systems with less than 256 processors, their primary focus being on capturing the system-related factors such as clock speed, network bandwidth via a multiplicative effects model. Barnes et al. [28] used linear regression techniques on total execution time and separately on communication and computation times. Their results show high median error on 1,024 cores for some applications. We show that interpolation techniques and a few machine learning techniques result in low median errors.

MuMMI [29] is an end-to-end automatic multiobjective modeling framework. It requires that training and testing points come from the same kernel or application. It uses linear correlation for input selection and a linear model to capture the relationship between PAPI counters and performance, power, and energy. In [30], six supervised learning methods were used to learn the relationship between hardware counters, source code transformation parameters (tiling, parallelization, etc.) and performance. It is a semi-automatic approach because input and model selection are manual. In [31], kernel-specific surrogate models, built by using artificial neural networks, were used to model the relationship between compiler transformation parameters and objectives such as power draw, execution time, and energy usage of HPC kernels.

Research in model-guided autotuning has focused on developing online empirical models for performance [32]–[34]. In [35], the authors developed online models for several scientific kernels on multicore architectures. In [36], the authors adopted boosted regression trees for obtaining online models for a GPU implementation of an image-filtering kernel. We demonstrate the usage of our techniques on different applications on different systems. The authors in [37] modeled the parallel I/O library PIDX using various machine learning techniques and regression. We focus on overall application performance, and our models can be enhanced by using their performance models for I/O. Lee et al. [38] studied artificial neural networks and polynomial regression for two applications on two systems. We extensively evaluated nine different learning algorithms for four different types of applications and benchmarks on three different types of systems. Recently, Marathe et al. [39] used transfer learning within deep neural networks to combine training data obtained from application runs collected at a smaller scale, with limited training data collected at a larger scale, to predict performance in surrogate-model-based autotuning. Thiagarajan et al. [40] developed a semi-supervised learning method to adaptively sample and find high quality configurations using limited number of parameter evaluations.

III. Machine-learning-based modeling

A class of machine learning (ML) approach that is particularly suitable for empirical performance modeling is supervised learning. This approach models the relationship between the output variable (such as runtime) and one or more independent input variables (such as application, software, and hardware parameters). The key aspect of this approach is collecting training data (x_i, y_i), for $x_i \in \mathcal{T} \subset \mathcal{D}$, where $\mathcal{T}$ is a set of training points, $\mathcal{D}$ is the full data set, and x_i and $y_i = f(x_i)$ are inputs and its corresponding output y_i, respectively. Here, x_i is a multidimensional tuple. The function f that maps the inputs to the output is typically unknown and hard to derive analytically. The goal of the supervised learning approach is to find a surrogate function h for f such that the difference between $f(x_i)$ and $h(x_i)$ is minimal for all $x_i \in \mathcal{T}$.

Many supervised learning algorithms exist in the ML literature. Based on the internal functionality, we can group them as follows: regularization, instance-based methods, recursive partitioning, kernel-based methods, artificial neural networks, bagging, and boosting methods. Typically, the best method depends on the properties of the data, such as volume of data, variety of data, and speed required for processing the data. We selected several algorithms to cover different groups. In the rest of the section, we provide a high-level overview the ML methods considered in our study.

Multivariate linear regression (`lm`) tries to find $h(x)$ by fitting a linear equation $h(x) = c + \sum_{i=1}^{M} \alpha^i \times x^i$, where c is a constant and α^i is the coefficient of the input x^j. Appropriate values of (c, α) are obtained by the method of least squares, which minimizes the sum of the squared deviations between the y_i and $h(x_i)$ in $\mathcal{T}$.

Ridge regression (`rg`) [41] is a regularization algorithm that addresses the overfitting problem faced by linear regression. Overfitting occurs when the model becomes sensitive to even small variations in the training set output and loses prediction accuracy on the testing and validation set. To avoid this problem, `rg`, in addition to minimizing the error between predicted and actual observations, penalizes the objective by a factor α of the input coefficients. Here, $\alpha \geq 0$ is a user-defined parameter that controls the tradeoff between minimizing the error and minimizing the sum of the square of the coefficients.

k-nearest-neighbor regression (knn) [42] belongs to the class of instance-based methods, where the model is not built explicitly during training phase but determined only during prediction. When a prediction is required for a testing point x^*, knn finds k nearest training points and returns the mean of the corresponding k outputs as the predicted value. Typically, k and the distance metric are the parameters, which are specific to the learning task and determined by the user.

Support vector machine (svm) [43] is a kernel-based method in which the input space is projected onto a higher-dimensional feature space by using a kernel function and linear regression is performed in that space. It tries to minimize ϵ-insensitive loss function that ignores errors within the ϵ-insensitive zone of the observed values and measures only the errors that lie outside this zone. This minimization problem can be formulated as a convex quadratic optimization problem, which can be solved to find the global optimum by using efficient optimization algorithms. The prediction accuracy of svm depends on a good choice of ϵ, kernel type, and their parameters.

Gaussian process regression (gp) [44] is a kernel-based method that assumes that the training data comes from a multivariate Gaussian distribution with mean vector $\mu(x)$ and covariance matrix $\sum(x)$ given by $\sum_{ij} = k(x^i, x^j)$, where k is a user-defined positive definite kernel function. The training process consists of optimizing the hyperparameters of the kernel k by maximizing the log-marginal-likelihood. When the kernel k determines that x^i and x^j are similar, then the predicted output at those points will be similar as well.

Decision tree (dt) [45] regression belongs to the class of recursive partitioning methods. This algorithm recursively splits the multidimensional input space of training points into a number of hyperrectangles such that inputs with similar outputs fall within the same rectangle. At each split, the dt selects among all inputs the one that minimizes the mean squared error. The splits gives rise to a set of if-else rules that can be represented as a decision tree. For each hyperrectangle, a constant value is assigned, typically an average over the output values that lie within the given hyperrectangle. Given a new testing point x^*, the algorithm uses the if-else rule to find the leaf and returns the corresponding constant value as the predicted value.

Random forest (rfr) [46] is a bagging approach that builds a number of decision trees (say, nt), each on random subsamples of the original dataset, and aggregates their individual predictions to form a final prediction. At each split, rfr considers only a random subset of inputs and selects the best one for the split based on the mean squared error.

Extremely randomized trees (etr) [47] differ from rfr in the way in which the splits are computed. As in rfr, a random subset of inputs is considered for the split; but instead of looking for the best mean squared error, the values are drawn at random for input, and the best of these randomly-generated values is picked as the splitting rule.

Gradient boosting regression (gbr) [48] is similar to rfr but with the following differences. The nt trees are built sequentially on a random subset of the training points. Each tree is generated with depth dpt, and its leaves have at least $\min_o$ observations. At the bth iteration, a tree model is built to minimize the prediction error of the $(b-1)$th model. The key idea is that the residuals of the $(b-1)$th model are used as the negative gradient of the squared error loss function being minimized. Similar to gradient-descent algorithms, gbr generates a new model at the bth iteration by adding the bth tree that fits the negative gradient to the $(b-1)$th model. The bth model is multiplied by a parameter $0 < \lambda \leq 1.0$ to control the bias-variance tradeoff.

eXtreme Gradient Boosting (xgb) [49] is a high-performing gradient-boosting optimized software framework and implementation, which is used widely by data scientists to achieve state-of-the-art results on many ML challenges such as Kaggle and KDDCup. The key advantage of xgb over other boosting algorithms is its scalability. This stems from several careful algorithmic optimizations—a novel approximate tree learning algorithm, an efficient procedure to handle training point weights in approximate tree learning, and system level optimization—computed out-of-core to process data that is too large to fit in main memory, and cache-aware learning to improve speed and scalability.

Deep neural networks (dnn) [50] belong to the class of artificial neural networks. They are characterized by a stack of layers organized in a hierarchical way. Each layer is composed of a number of units, each of which contains a nonlinear activation function. The training points are presented to the network through the input layer, which communicates with the stacked hidden layers where the computations are performed via a system of weighted connections. The last hidden layer is connected to an output layer from which the predicted output is obtained. The training phase consists of modifying the weights of the connections in the network to minimize an error function that measures prediction error. This is typically done by a back propagation method that computes the gradient of the error function with respect to all the weights in the network and uses it in a gradient-based optimization method to update the weights.

IV. Experimental Evaluation

In this section, we describe our target HPC platforms, application benchmarks, implementation details of the ML methods, evaluation metrics, and prediction accuracy results.

A. HPC platforms

We used HPC platforms with diverse architectures and network topologies for generating training data for the benchmark applications. We experimented on four different systems: two IBM Blue Gene/Q systems and two Cray systems. We used Mira and Vesta at the Argonne Leadership Computing Facility at the Argonne National Laboratory [9] and Edison and Hopper at the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory [10], [11]. While Hopper is a retired platform, the application runs were obtained from our previous experiments, which we reused for

this study. The systems configurations are listed in Table I. The last column lists the maximum number of cores used for our evaluation on these systems. The Blue Gene/Q systems have 16 in-order processor cores per node with a 2-level cache hierarchy and a 5D torus interconnect topology (∼2 GB/sec peak MPI bandwidth). Vesta has 16 I/O nodes for every 512 compute nodes, whereas Mira has 4 I/O nodes for every 512 compute nodes. Apart from the difference in number of I/O nodes, Vesta (a test and development system) also has a smaller queue waiting time than Mira (a production system has). The Cray systems have two 12-core processors per node. Hopper has a 2-level cache hierarchy, and Edison has a 3-level cache hierarchy. Hopper has a Cray Gemini network (∼5.8 GB/sec peak MPI bandwidth) and has a complex wiring to realize the 3D torus [11]. Edison has a Cray Aries high-speed interconnect with Dragonfly topology (∼8 GB/sec peak MPI bandwidth).

TABLE I
MACHINE CONFIGURATIONS

Name	Processor	Interconnect Topology	Maximum # Cores
Mira (Blue Gene/Q)	Power BQC 1.6 GHz	5D torus	131,072
Vesta (Blue Gene/Q)	Power BQC 1.6 GHz	5D torus	16,384
Edison (Cray XC30)	Intel Ivy Bridge 2.4 GHz	Aries with dragonfly	1,728
Hopper (Cray XE6)	AMD MagnyCours 2.1 GHz	Gemini with 3D torus	12,000

On Vesta, we performed experiments on 32–1024 nodes, increasing the node count by a power of 2. We used 512–8192 nodes on Mira, increasing the node count by a power of 2. On Edison, we used results from 32 different process counts between 24–1728, somewhat uniformly spaced. On Hopper, we collected results from 36 different process counts between 24 and 12,000, somewhat evenly spaced. We measured performance for a small number of time steps, which is a commonly used approach [51]. However, we incurred large data collection times, including queue wait times (ranging from a few minutes to days), especially for large node counts.

B. Benchmarks

We used three miniapplications from the Mantevo benchmark suite [6] and a large-scale molecular dynamics simulation for evaluation. In all these applications, we modeled runtime as a function of application-specific inputs. Next, we briefly describe them.

1) miniMD: miniMD is a molecular dynamics miniapplication developed at Sandia National Laboratories. It mimics calculations such as the force computations in typical molecular dynamics application. The algorithms and implementation used closely mimic these operations as performed in LAMMPS. It performs simulation of a Lennard-Jones system and does spatial decomposition across MPI processes. The output (predicted variable) in this case is the execution time, and the input is the number of MPI processes and the number of atoms. We experimented with 13,500–1,21,94,500 atoms on 32–1,024 nodes of Vesta, 16 ranks per node (i.e. 512–16384 cores), resulting in 288 data points; we refer to this dataset as miniMD (Vesta). On Hopper, we used 2,048–1,57,77,248 atoms for simulation on 24–12,000 cores, resulting in 1,620 data points; we refer to this dataset as miniMD (Hopper). We used the number of atoms and the number of processes as inputs.

2) miniAMR: miniAMR is a miniapplication designed to study adaptive mesh refinement (AMR) codes at scale [26]. It does a 3D stencil calculation on a unit cube initial computational domain, which is divided into blocks. With AMR, the blocks can represent different levels of refinement in the larger mesh. MiniAMR allows the user to specify a number of parameters, including refinement levels, maximum number of blocks, refinement frequency, and number of processes in each direction (x, y, z). We varied the block size from 2 to 12 in each direction (x, y, z) and used refinement level 4. The refinement frequency was every 10 steps. We used the example problem of two moving spheres provided with the benchmark. The output (predicted variable) in this case is the execution time, and the input is the number of MPI processes in x, y, z and the number of blocks in x, y, z. We experimented on 32–1,024 nodes on Vesta (72 data points; we refer to this dataset as miniAMR (Vesta)); we also experimented on 512–4,096 nodes of Mira, 4 ranks per node, (864 data points; we refer to this dataset as miniAMR (Mira)). We used six parameters as input to our models–the number of processes in x, y, z and the block size in x, y, z. Instead of the total number of processes, we used the x, y, z process decomposition as inputs because they can capture the interaction and effect of these parameters on performance.

3) miniFE: MiniFE is a miniapplication for unstructured implicit finite element codes. It is intended to be the best approximation to an unstructured implicit finite-element or finite-volume application, but in 8,000 lines or fewer [6]. It implements a couple of kernels representative of implicit finite-element applications [26]. It assembles a sparse linear system from the steady-state conduction equation on a brick-shaped problem domain of linear 8-node hex elements. It then solves the linear system using a simple unpreconditioned conjugate-gradient algorithm. We experimented on 512–8192 nodes on Mira, 16 ranks per node (i.e. 8,192–131,072 cores), for domain size of 100–600 in each dimension (x, y, z), a total of 6,655 data points; we refer to this dataset as miniFE (Mira). This results in skewed input configurations such as $100\times600\times600$. On Edison, we evaluated on 24 – 1728 cores for domain size of 100 – 450 in each dimension, a total of 15360 data points; we refer to this dataset as miniFE (Edison). We used the number of processes and the grid sizes in x, y, z as inputs.

4) LAMMPS: The Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [7] is a classical molecular simulation code. It uses spatial decomposition of simulation domain for parallelism and provides flexibility to change the

input size of a problem via a simple replicate command in the input script. The problem investigated here is a box of water molecules solvating two types of ions. We evaluated our performance prediction models for problem sizes of 8 million–0.1 billion atoms on 64–1024 nodes of Vesta, 16 ranks per node (50 data points); we refer to this dataset as LAMMPS (Vesta); We did the same for the problem sizes of 12,544–50 million atoms on 128–1,024 nodes of Mira, 16 ranks per node (500 data points); we refer to this dataset as LAMMPS (Mira). We used the number of processes, the number of atoms, and the box size in x, y, z as inputs.

The number of dimensions for miniMD, miniAMR, miniFE, and LAMMPS is 2, 6, 4, and 4 respectively. In total, we had eight application datasets. The miniapps do not incur significant I/O activities. I/O was turned on for LAMMPS; however, we measured the compute performance separately and present the results for modeling the same.

C. Implementation

All the ML methods described in Section III are implemented in Python (Intel distribution, version 3.6.3) using the scikit-learn library (version 0.19.0). We used default parameters provided by scikit-learn library for all algorithms. For preprocessing, we applied `MinMaxScaler` transformation to scale each input and output between 0 and 1 before training and applied the inverse transformation after prediction so that evaluation metrics are computed on the original scale. For `rfr`, `etr`, and `xgb`, scikit-learn has different nt (number of trees) values as default values. However, we set the number of trees to $1,000$ in all these methods to avoid inconsistency and consequent prediction accuracy differences.

For `dnn`, we used Keras (version 2.0.8), a high-level neural networks Python library that runs on the top of the TensorFlow library (version 1.3.0). We note that there is no default or general-purpose network configuration. In fact, designing an appropriate network for a given task is often difficult and an open research question for many learning tasks. We carried out an exploratory study and developed a multilayered feed-forward neural network as shown in Figure 1: the input layer of size $|X|$ (in Figure 1, this is set to 10 for illustration) is connected to a block that comprises a dense hidden layer with 512 units (dense_1), a dropout layer (dropout_1) with a rate of 0.2 to prevent overfitting, and a rectified linear activation function (not shown in the figure). This block configuration is repeated twice (dense_2, dropout_2, dense_3, and dropout_3), where the output of the previous block is given as the input for the next block. The last layer is the output layer of size 1 that predicts the value for an unknown input x^*. We trained the `dnn` with the objective of minimizing the mean squared error using the gradient-based Adam optimizer with a batch size of 512 and a stopping criterion of 100 epochs.

D. Evaluation metrics

We used two metrics to evaluate to measure how close the predictions are to the observed values, and we compared the effectiveness of the ML methods under study.

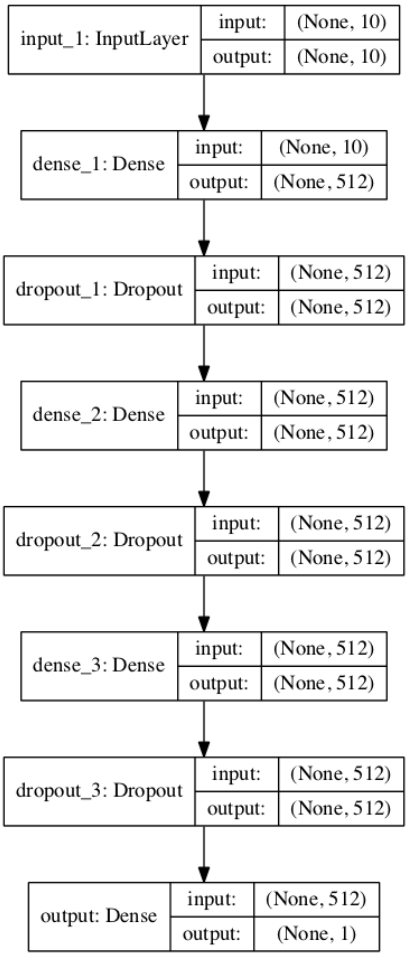

Fig. 1. Network architecture used for the `dnn`.

The first metric is the median absolute percentage error (MedAPE) given by the median of $100 \times \frac{|y^i - \hat{y}^i|}{y_i}\%$ for $i \in 1, \ldots, n$, where y_i and $\hat{y}_i$ are observed and predicted values of the test data point i. Since the absolute percentage error can be sensitive to outliers and the magnitude of the application runtime, we rely on the coefficient of determination (R^2) between the observed values and the fitted values. This metric is typically used to assess the goodness of the model fit. The maximum value of R^2 is 1, which indicates that the model predictions perfectly fit the data.

We use cross-validation to assess the robustness of our ML methods and to test the generalizability of the model on an independent data set. From a given application dataset, we generated ten 20/80 cross-validation datasets, where each dataset comprises of a randomly sampled 20% of the data for training the model and the remaining 80% for testing. For each application, we ran each ML method on the ten cross-validation datasets. We computed the evaluation metrics on each cross-validation dataset and report the aggregated value. Note that in a typical ML setting, 80/20 cross-validations are used for model evaluations. Since our goal was to evaluate these models under a limited number of training points, we

use a 20/80 cross validation.

E. Results

In this section, we present results from benchmarking ML methods for performance modeling with the four scientific application datasets.

1) Impact of ML method complexity and feature engineering on accuracy: To evaluate the accuracy of the ML methods, we considered two datasets from each application dataset. In the first set, we used the raw inputs (total number of processes, node count, and application-specific inputs such as the number of atoms, grid sizes, and box sizes as described in Section IV-B). In the second set, we generated new nonliner input features that capture the domain decomposition and scalability of the application—a process called feature engineering (FE). These features include the ratio of the application problem size and the number of number of processes, inverse of the number of processes, and binary logarithm of number of processes. We combined these new inputs and the raw inputs. We refer to the two datasets as No-FE and FE, respectively.

We ran training and inference on a NVIDIA DGX-1 platform: Dual 20-Core Intel Xeon E5-2698 v4 2.2 GHz with 8x Nvidia P100 GPUs with 512 GB of memory. All ML methods except `dnn` used only the host CPU processors; the `dnn` leveraged GPUs.

The results are shown in Figure 2, where each box-whisker[1] is obtained from 10 MedAPE values (1 application dataset $\times$ 10 folds). On the No-FE dataset, we observe that the algorithmic complexity of the ML method has a significant impact on the accuracy. The ML methods with better learning capabilities such as `gp`, `etr`, `gbr`, `xgb`, and `dnn` outperform other ML methods: 95% quantiles of the MedianAPE distribution obtained by these methods are less than 20% on six of eight application datasets.

The absolute percentage error values are affected by low accuracy on large runtime values that correspond to small node count application runs. Therefore, for a given application, dataset variant, method, and a fold, we computed the R^2 value from the observed and predicted values of the testing dataset. We then aggregated the R^2 values across application datasets and analyzed the distribution. The results are shown in Figure 3, where each box-whisker plot is obtained from 80 R^2 values (8 application dataset $\times$ 10 folds). We observe that `gbr`, `xgb`, and `dnn` obtain R^2 values between 0.7 (worst case) and 1.0 (best case), which are significantly higher than the values obtained by other ML methods. The ineffectiveness of `lm` and `rg` show that there is a significant nonlinearity in the input-output relationship.

On the FE data set, R^2 values of the relatively simple ML methods (`lm`, `rg`, `knn`, `dt`, `svm`, and `gp`) improve significantly. In particular, `lm` using the nonlinear features obtains R^2 values between 0.95 (worst case) and 1.0 (best case). These results clearly show that feature engineering is required for the adoption of simple ML methods. For `etr`, `gbr`, `xgb`, and `dnn`, the improvements with FE datasets are small because these methods are capable of automatically learning the necessary features from the training data without feature engineering. The accuracy of these methods is already high on No-FE datasets, and feature engineering results in only minor improvements; in particular, it improved the worst-case prediction accuracy.

2) Impact of hardware platforms: We benchmarked the ML methods on different hardware platforms and studied their effects on accuracy, training, and inference time. In addition to the default NVIDIA DGX-1 platform with Tesla V100 (we refer to this platform as V100), we used two platforms. The first platform is a 2.4 GHz Intel Haswell E5-2620 v3 processor (6 cores per CPU), 384GB RAM, with a NVIDIA Tesla K80 (two K40 GPUs) per node, and 24 GB GPU RAM per node (12 GB per GPU). We used only one GPU for training and inference (consequently, we refer to this platform as K40). Second platform is a 3.4 GHz Intel Xeon E5-2687W processor (8 cores per CPU), 64GB RAM, with a NVIDIA Tesla P100, 16 GB GPU RAM (we refer to this platform as P100). For each application dataset, we used 10 folds of the No-FE dataset, each with randomly sampled 20% training and 80% testing points.

The results are shown in Figure 4. Each box-whisker plot is obtained from 80 (8 application datasets $\times$ 10 folds) values. The box-whisker plots on R^2 values show that the hardware platforms did not affect the accuracy in a significant way. To enable aggregation and comparison across applications, we normalized the training and testing time. For a given application dataset fold and ML method, we computed the ratio of the total training (inference) time and the number of training (inference) data points. We observe that the adoption of different hardware platforms does not result in significant differences in normalized training and inference times. On the other hand, the complexity of the ML method has a significant impact on the normalized training and inference times. The normalized training times of `lm`, `rg`, `knn`, `dt`, and `svm` range between 10^{-6}s and 10^{-4}s; `etr`, `rfr`, `gbr`, and `xgb` range between 10^{-4}s and 10^{-1}s. Even with the adoption of GPUs, `dnn` is relatively more computationally expensive than most of the shallow learning methods. This cost is due to the $O(n_1^3)$ time complexity, where n_1 is the matrix-matrix multiplication kernel size (in our case 512). The bagging and boosting methods typically have a time complexity of $O(N\ log\ N)$ in the training set size N and linear in the input feature size. The observed trends are similar on the normalized inference time; however, the inference is faster than training by approximately 1 to 2 orders of magnitude.

3) Impact of training dataset size: In this section, we study the impact of the size of training dataset on the prediction accuracy. We compare the accuracy of the ML methods with the default 20% training data to the accuracy of the ML methods when trained on 1%, 5%, 10%, and 30% data and

[1]The distribution of the accuracy values is shown using box and whisker plots: the ends of the box are the 25% (lower) and 75% (upper) quantiles of the distribution, the vertical notch inside the box is the median of the distribution, the whiskers are the two lines outside the box, whose ends represent 5% and 95% of the distribution, and the dots mark the outliers of the distribution.

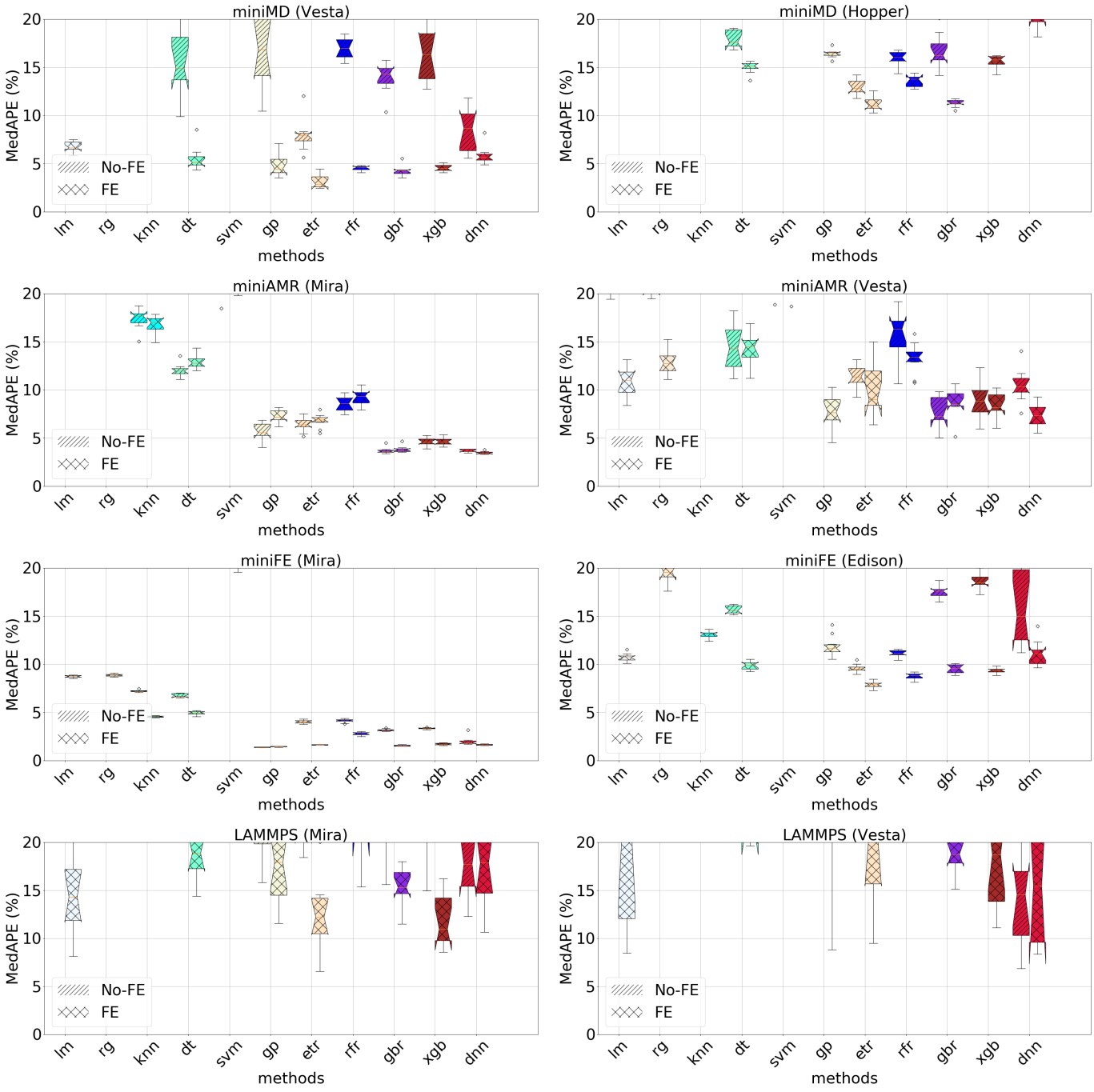

Fig. 2. Box-whisker plots showing the distribution of the MedianAPE values obtained by the ML methods on the application datasets. Each ML method is evaluated on raw (No-FE) and feature-engineered (FE) dataset. The ML methods in the plots are (roughly) ordered based on their ability to learn nonlinear relationship between the inputs and output from the training data. A missing box plot for a particular ML method means that all the 10 MedianAPE values are greater than 20%.

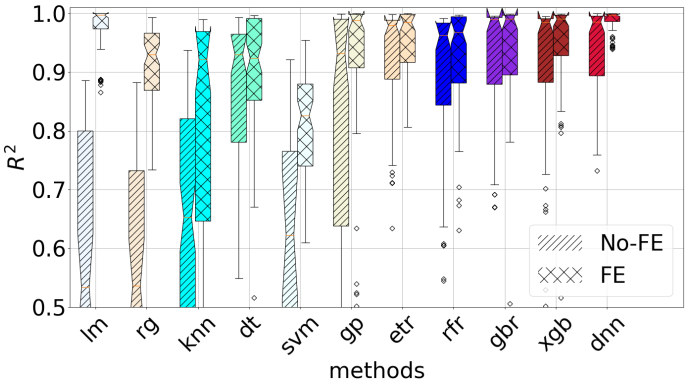

Fig. 3. Box-whisker plots showing the distribution of the prediction accuracy (R^2) values obtained by the ML methods. Each ML method is evaluated on a raw (No-FE) and a feature-engineered (FE) dataset. The ML methods are (roughly) ordered based on their ability to learn nonlinear relationship between the inputs and output from the training data.

tested on 99%, 95%, 90%, and 70% data, respectively.

The results are shown in Figure 5. The box-whisker plot for each method is obtained from 80 R^2 values (8 application datasets $\times$ 10 folds). We observe that increasing the number of training points increases the prediction accuracy for `knn`, `dt`, `gp`, `etr`, `rfr`, `gbr`, `xgb`, and `dnn`. Nevertheless, `lm`, `rg`, and `svm` did not take advantage of the large number of training points. Comparing these results with those in Figure 3, we conclude that feature engineering is more important than the training set size for these methods. We observe a significant increase in the R^2 values when increasing the training points from 1% to 5%, and to 10%; nevertheless, the improvement from 20% to 30% is small.

4) Transfer learning: A key advantage of `dnn` is transfer learning (TL), where a network trained on one task can be adapted to a similar task with limited training data. Our hypothesis is that a network trained on an application from HPC platform P_1 can be adapted to predict the application performance on a different HPC platform P_2 with a few training points of the application run on platform P_2.

To test our TL hypothesis, we used the following setting. For a given application, we trained the `dnn` (see Figure 1) with 20% training points on platform P_1. We froze the weights of dense_1 and dense_2 layers (dropout layers do not have weights), used 1% training data of the same application from the platform P_2, and retrained the `dnn`, where the weights of the (dense_3) are adjusted. We used the `dnn` to predict the 99% testing data on the platform P_2. We compared the prediction accuracy of this `dnn` with TL to a `dnn` that used only 1% training data from the platform P_2.

The results are shown in Figure 6. The `dnn` with TL for miniFE (Mira) used 20% of the miniFE-Edison dataset and 1% of the miniFE-Mira dataset for training. Similarly, the `dnn` with TL for miniFE (Edison) used 20% of the miniFE-Mira dataset and 1% of miniFE-Edison dataset. Each box-whisker plot is obtained from 10 R^2 values (repeating the experiments

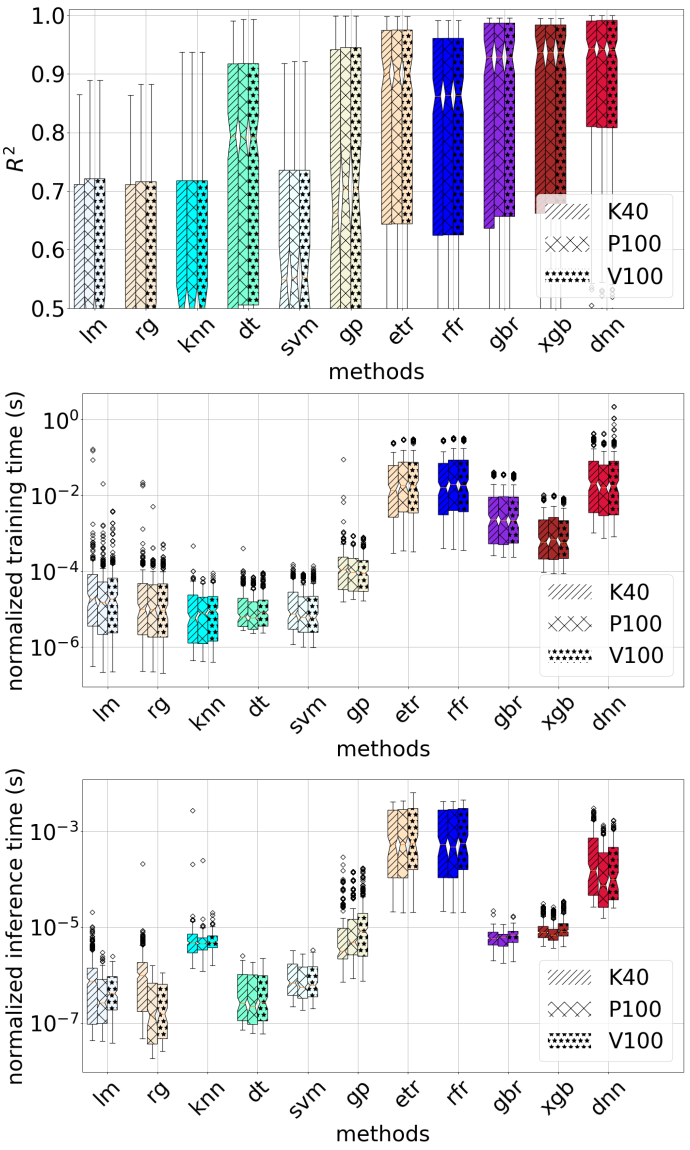

Fig. 4. Box-whisker plots of accuracy (R^2 values), normalized training, and inference times of the ML methods on different hardware platforms.

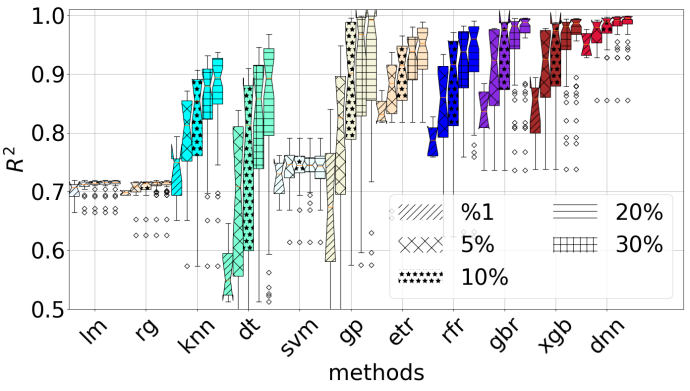

Fig. 5. Impact of training dataset size on prediction accuracy.

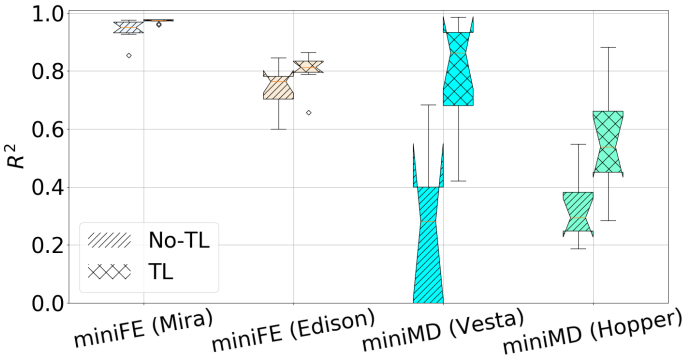

Fig. 6. Effectiveness of transfer learning (TF) with `dnn`.

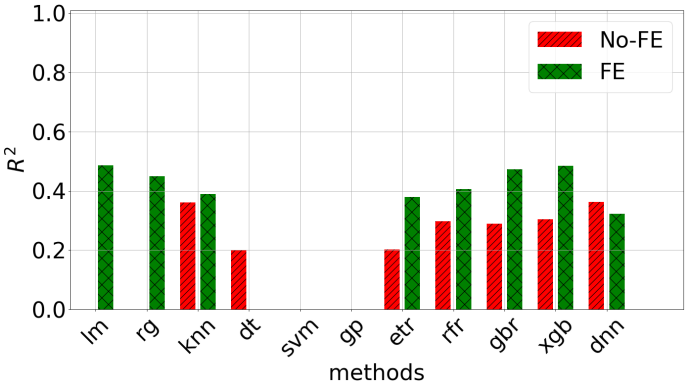

Fig. 7. Bar plot showing the extrapolation accuracy of the ML methods on the miniFE datasets obtained on Edison.

10 times using 10 folds). From the results, we observe that the `dnn` with TL significantly outperforms the `dnn` without TL. For miniMD, the use of TL increases the median R^2 from 0.4 to 0.9 (Vesta) and from 0.3 to 0.5 (Hopper). The best case R^2 values of 0.99 and 0.85 indicate that when training points are selected appropriately (in our case, although the folds are sampled randomly, this particular fold got lucky), one can obtain high accuracy even under a limited-data regime, motivating the need for systematic sampling methods such as active learning. For miniFE, the improvements in R^2 values are significant in a statistical sense. In particular, we observe that the use of TL significantly reduces the worst case R^2 value on both Mira and Edison.

5) Extrapolation results: We studied the effectiveness of the ML methods for extrapolating the runtimes on higher node counts using data from smaller node counts. Note that although ML methods are capable of generalizing the function inside the subspace spanned by the training samples but outside that subspace, their prediction power reduces drastically.

For extrapolation experiments, we considered miniFE datasets on Edison. We used data for 24–1,152 processes as the training set (11,520 data points) and we predicted on 1,224–1,728 processes (3,840 data points). Note that because of this split, only one fold is feasible. We used No-FE and FE versions of the datasets.

The results are shown in Figure 7, where we observe that R^2 values obtained by all the methods are less than 0.5. The missing bars in the plot are due to large errors that resulted in R^2 values that are not meaningful. These results show that extrapolation using ML models outside the training subspace will not work for all settings and it requires deeper training subspace characterization. We also observed that the use of FE dataset resulted in a significant improvement in R^2 values. In particular, `lm` and `rg` obtain R^2 values similar to that of `xgb` and `gbr`.

V. Conclusions and Future Work

Accurate and fast performance models are useful for job scheduling at computing sites where different types of applications are executed on a daily basis. Empirical performance modeling is an effective alternative when the complex interactions of the system and application are hard to capture analytically. A class of ML algorithms proves to be efficient in performance modeling. To that end, we evaluated a broad range of ML methods for modeling the performance of a representative set of scientific applications on diverse HPC platforms with three types of network topologies. We select representative ML methods from different classes and benchmarked them for modeling the performance of scientific applications on leadership-class systems. The conclusions from our benchmark studies are as follows:

- The sophisticated ML methods such as bagging, boosting, and deep neural networks outperform all other ML methods when the raw inputs were used for training. Simple ML methods such as linear regression and ridge regression require feature engineering to achieve high prediction accuracy.
- The algorithmic and computational complexities of the ML methods have a significant impact on accuracy, model training and inference times. For the given ML method, the modern hardware platforms that we tested do not produce significant accuracy, training-, and inference-time differences.
- Experiments on training set size showed that bagging, boosting, and deep neural networks leverage large training datasets and produce better accuracy. Moreover, for the considered applications, 20% of the randomly sampled application data is sufficient to reach high accuracy.
- The results on transfer learning showed that a deep neural network trained on an application data from one HPC platform can be reused to improve the prediction accuracy of the network for the same application on a different HPC platform with limited data.
- While ML methods worked well for interpolating performance, extrapolations to large node counts from small node counts are hard and require careful consideration. The results showed that feature engineering that incorporates HPC/application knowledge will be crucial for extrapolation.

Our future work will include design and development of active learning mechanisms to select training points in a guided

manner; improve transfer learning using analytical models, application and hardware knowledge; and improve the extrapolation performance by using training subspace characterization and transfer learning.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research program under contract number DE-AC02-06CH11357. We gratefully acknowledge the computing resources at the Argonne Leadership Computing Facility and National Energy Research Scientific Computing Center.

REFERENCES

[1] L. G. et al., "Tightly coupled atomistic-continuum simulations of brain blood flow on petaflop supercomputers," *Computing in Science Engineering*, vol. 14, no. 6, pp. 58–67, 2012.

[2] P. M. et al., "A divide and conquer strategy for scaling weather simulations with multiple regions of interest," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[3] ——, "Optimal scheduling of in-situ analysis for large-scale scientific simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[4] L. e. a. M, "Performance modeling of in situ rendering," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[5] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for performance modeling and prediction," in *Supercomputing, ACM/IEEE 2002 Conference*, 2002.

[6] M. H. et al., "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[7] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[8] R. Barrett, P. Crozier, D. Doerfler, M. Heroux, P. Lin, H. Thornquist, T. Trucano, and C. Vaughan, "Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications," *Journal of Parallel and Distributed Computing*, vol. 75, pp. 107 – 122, 2015.

[9] "Computing resources at argonne leadership computing facility," https://www.alcf.anl.gov/computing-resources.

[10] "Edison," http://www.nersc.gov/users/computational-systems/edison.

[11] "Hopper," http://www.nersc.gov/users/computational-systems/retired-systems/hopper.

[12] S. Pllana, I. Brandic, and S. Benkner, "Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art," in *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*, 2007, pp. 279–284.

[13] R. Allan and A. Mills, "Survey of HPC performance modelling and prediction tools," *Science and Technology*, 2010.

[14] D. J. Kerbyson, K. J. Barker, and K. Davis, "Analysis of the weather research and forecasting (WRF) model on large-scale systems," in *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007*, 2007, pp. 89–98.

[15] D. J. Kerbyson and P. W. Jones, "A performance model of the parallel ocean program," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 3, pp. 261–276, 2005.

[16] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, "Exasat: An exascale co-design tool for performance modeling," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 2, pp. 209–232, 2015.

[17] X. Wu and V. Taylor, "Performance modeling of hybrid MPI/OpenMP scientific applications on large-scale multicore supercomputers," *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1256–1268, 2013.

[18] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 336–346, 2006.

[19] R. Iakymchuk and P. Bientinesi, "Modeling performance through memory-stalls," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 2, pp. 86–91, 2012.

[20] "Dimemas," http://www.bsc.es/computer-sciences/performance-tools/dimemas.

[21] Z. Tong, S. Pakin, M. Lang, and X. Yuan, "Fast classification of mpi applications using lamport's logical clocks," in *Proceedings of the 30th IEEE International Parallel and Distributed Computing*, 2016.

[22] X. Wu and F. Mueller, "ScalaExtrap: Trace-based Communication Extrapolation for SPMD Programs," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, 2011, pp. 113–122.

[23] L. Carrington, M. A. Laurenzano, and A. Tiwari, "Inferring large-scale computation behavior via trace extrapolation," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013, pp. 1667–1674.

[24] A. Wong, D. Rexachs, and E. Luque, "Parallel application signature for performance analysis and prediction," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 2009–2019, July 2015.

[25] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely, *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*, 2009, ch. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications, pp. 135–148.

[26] "Characterization of the DOE Mini-apps," http://portal.nersc.gov/project/CAL/designforward.htm.

[27] J. Delgado, S. Sadjadi, M. Bright, M. Adjouadi, and H. Duran-Limon, "Performance prediction of weather forecasting software on multicore systems," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1–8.

[28] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08, 2008, pp. 368–377.

[29] X. Wu, C. Lively, V. Taylor, H.-C. Chang, C.-Y. Su, K. Cameron, S. Moore, D. Terpstra, and V. Weaver, "MuMMI: Multiple metrics modeling infrastructure," in *ACIS Int. Conf. Softw. Engin., Artif. Intel. Network. Parallel/Distributed Comp. (SNPD)*, 2013, pp. 289–295.

[30] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," *Int. J. Parallel Programming*, vol. 41, no. 5, pp. 704–750, 2013.

[31] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling power and energy usage of HPC kernels," in *IEEE Int. Conf. Par. Distrib. Proc. Symp. Workshops (IPDPSW12)*, 2012, pp. 990–998.

[32] G. Fursin *et al.*, "MILEPOST GCC: Machine learning based research compiler," in *GCC Summit*, 2008.

[33] A. Magni, C. Dubach, and M. F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *Proc. Int. Conf. High Perf. Comp. Networking Storage Anal.*, ser. SC '13, 2013.

[34] O. Spillinger, D. Eliahu, A. Fox, and J. Demmel, "Matrix multiplication algorithm selection with support vector machines," Master's thesis, EECS Department, University of California, Berkeley, May 2015.

[35] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *IEEE Int. Conf. Data Engineering (ICDE'09)*, 2009, pp. 592–603.

[36] J. Bergstra, N. Pinto, and D. Cox, "Machine learning for predictive auto-tuning with boosted regression trees," in *Innovative Parallel Computing (InPar'12)*. IEEE, 2012, pp. 1–9.

[37] S. e. a. Kumar, "Characterization and modeling of PIDX parallel I/O for performance optimization," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 67:1–67:12.

[38] B. C. e. a. Lee, "Methods of inference and learning for performance modeling of parallel applications," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07, 2007, pp. 249–258.

[39] A. Marathe, R. Anirudh, N. Jain, A. Bhatele, J. Thiagarajan, B. Kailkhura, J.-S. Yeom, B. Rountree, and T. Gamblin, "Performance modeling under resource constraints using deep transfer learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 31.

[40] J. J. Thiagarajan, N. Jain, R. Anirudh, A. Gimenez, R. Sridhar, A. Marathe, T. Wang, M. Emani, A. Bhatele, and T. Gamblin, "Bootstrapping parameter space exploration for fast tuning," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: ACM, 2018, pp. 385–395. [Online]. Available: http://doi.acm.org/10.1145/3205289.3205321

[41] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.
[42] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, New York, 2006, vol. 1.
[43] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, 2004.
[44] C. E. Rasmussen, "Gaussian processes in machine learning," in *Advanced lectures on machine learning*. Springer, 2004, pp. 63–71.
[45] W.-Y. Loh, "Classification and regression trees," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 14–23, 2011.
[46] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[47] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.
[48] J. H. Friedman, "Stochastic gradient boosting," *Comput. Stat. Data Anal.*, vol. 38, no. 4, pp. 367–378, 2002.
[49] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," *arXiv preprint arXiv:1603.02754*, 2016.
[50] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[51] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2005.