Automated Performance Modeling of HPC Applications Using Machine Learning

Jingwei Sun, Guangzhong Sun, Shiyan Zhan, Jiepeng Zhang, and Yong Chen

Abstract—Automated performance modeling and performance prediction of parallel programs are highly valuable in many use cases, such as in guiding task management and job scheduling, offering insights of application behaviors, and assisting resource requirement estimation. The performance of parallel programs is affected by numerous factors, including but not limited to hardware, applications, algorithms, and input parameters, thus an accurate performance prediction is often a challenging and daunting task. In this study, we focus on automatically predicting the execution time of parallel programs (more specifically, MPI programs) with different inputs, at different scales, and without domain knowledge. We model the correlation between the execution time and domain-independent runtime features. These features include values of variables, counters of branches, loops, and MPI communications. Through automatically instrumenting an MPI program, each execution of the program will output a feature vector and its corresponding execution time. After collecting data from executions with different inputs, a random forest machine learning approach is used to build an empirical performance model, which can predict the execution time of the program given a new input. An instance-transfer learning method is used to reuse an existing performance model and improve the prediction accuracy on a new platform that lacks historical execution data. Our experiments and analyses of three parallel applications, Graph500, GalaxSee, and SMG2000, on three different systems confirm that our method performs well, with less than 20% prediction error on average.

Index Terms—Parallel computing, performance modeling, machine learning, model transferring.

1 Introduction

Performance modeling is a widely concerned problem in high performance computing (HPC) community. An accurate model of parallel program performance, particularly an accurate model for predicting execution time can yield many benefits. **First**, a performance model can be used for task management and scheduling, assisting the scheduler to decide how to map tasks to proper compute nodes [19], [37]. Therefore, the utilization of the entire HPC system can be improved. **Second**, the model can offer insights about application behaviors [22], [25], which helps developers understand the scaling potential and better tune applications. **Third**, the model helps HPC users to estimate the number of CPU cores they need [43], [44]. According to the predicted performance, users can consider the predicted computation time and estimated resources systematically, and then request a reasonable number of compute nodes and CPU cores from HPC systems.

Building an accurate performance model of parallel programs, however, is a very challenging task. Due to the variance and complexity of both system architectures and applications, the execution time of a parallel program is often with significant uncertainty. For example, numerous factors can affect the performance, including but not limited to hardware, applications, algorithms, and input parameters. It is especially difficult to build a general-purpose model that synthesizes all factors. In this paper, we focus on designing and developing a model, particularly for predicting the execution time of parallel programs, on an HPC cluster with different inputs and at different scales. We also focus on MPI programs as MPI is the de facto standard parallel programming model.

Previous studies have mainly introduced three types of methods: *analytical modeling*, *replay-based modeling*, and *statistical model*. An *analytical modeling* method [8], [26], [36], [40] has arithmetic formulas describing a parallel program performance and can offer a prediction of execution time quickly. However, this method needs extensive efforts of human experts with in-depth understanding of a particular HPC application (e.g. consider the time complexity analysis process of a parallel program). Since HPC applications have a wide range of domains, it is difficult to build an analytical model. Furthermore, it is challenging to generalize a model for various domains.

A *replay-based model* [21], [38], [44], [45] is built from historical execution traces, which contain detailed information about computation and communication of an HPC program. Through analyzing traces, a synthetic program can be automatically reconstructed for replaying behaviors of the original program and predicting its performance. However, the replay-based modeling usually requires large storage space to keep traces (ranging from hundreds of megabytes to tens of gigabytes for each run [9], [44]). Besides, a synthetic program can only represent one specific execution path of the original program, which also restricts the application of replay-based modeling.

A *statistical model* [9], [10], [14] uses machine learning techniques to fit the mapping function between perfor-

- *J. Sun, G. Sun, S. Zhan and J. Zhang are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. Email: sunjw@mail.ustc.edu.cn, gzsun@ustc.edu.cn, zsynew@mail.ustc.edu.cn, hitzjp@mail.ustc.edu.cn.*
- *Y. Chen is with Department of Computer Science, Texas Tech University, Lubbock, Texas, U.S.A. Email: yong.chen@ttu.edu.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2964767, IEEE Transactions on Computers

mance metrics and certain features. With sufficiently much training data, statistical models can make relatively accurate predictions of the performance, without requiring domain knowledge and human efforts. It is natural and convenient to use input parameters of an HPC program as features. However, important performance factors may not be explicitly covered by input parameters. For instance, it is difficult to automatically parse non-scalar inputs (e.g. files, matrices, and strings) for modeling without domain experts. Domain experts can determine a small number of scalar variables in the source code of a program as model features since these variables can directly expose the actual performance impact from inputs [9]. For applications that contain adaptive preprocess or auto-tuning [30], [31], some key features are dynamically decided at runtime. In this case, input parameters also cannot cover all performance features.

As a superset of input parameters, runtime features generally contain more complete and relevant information that affects performance. But effectively identifying an important subset of runtime features without domain knowledge is still challenging, since the number of possible runtime features usually is much larger than the number of input parameters. A statistical performance model should properly maintain its complexity, namely the number of features. Redundant features result in high expense and high generalization error, while a lack of important performance features makes the model useless.

In this paper, we propose a performance modeling and prediction method, which identifies runtime features that is highly relevant to performance, without requiring domain knowledge and human efforts. It automatically inserts instrumentation code, detects runtime features, and analyzes feature data. We adopt a lightweight instrumentation to reduce the overhead so that the instrumented program can be normally used for production executions and meanwhile continuously output feature data for constructing a more accurate model. The random forest regression approach [20] is used to predict execution time. With this approach, the performance model can fit the complex nonlinear mapping function between features and performance. It can also analyze the statistical importance of each feature and only reserve a subset of important features to further reduce the overhead of instrumentations. In addition to making a point prediction (e.g. scalar value of execution time), our model can also predict an interval of performance to show the performance uncertainty.

Statistical modeling requires repetitive executions of an application on a certain platform. For a new platform, it is difficult to build an accurate model due to the lack of historical execution data. It is the so-called cold start problem. To deal with this problem, we introduce an instance-transfer learning method that can reuse an existing performance model to assist in building an accurate model on a new platform with a small number of execution samples.

The main contributions of this study are summarized as follows:

- We propose a method to model and predict the performance of parallel programs (MPI programs) with different inputs. Our experiments with three different programs on three different HPC systems show that the average prediction error is less than 20%.
- We develop a tool to automatically analyze the syntax tree of an MPI program and instrument it. Thus we can detect domain-independent runtime features related to performance. These features are necessary to support the automated modeling and prediction with machine learning techniques.
- We design a strategy to automatically analyze and reorganize the runtime feature data of an MPI program using machine learning, thereby we can extract the factors that significantly affect the performance and reduce the overhead and storage demand from redundant instrumentations.
- We introduce an instance-transfer learning method that can transfer an existing performance model to adapt to a new platform. It only uses 5% of the execution samples from the new platform as the training set to predict the execution time of the remaining samples, and the average prediction error is less than 20%.

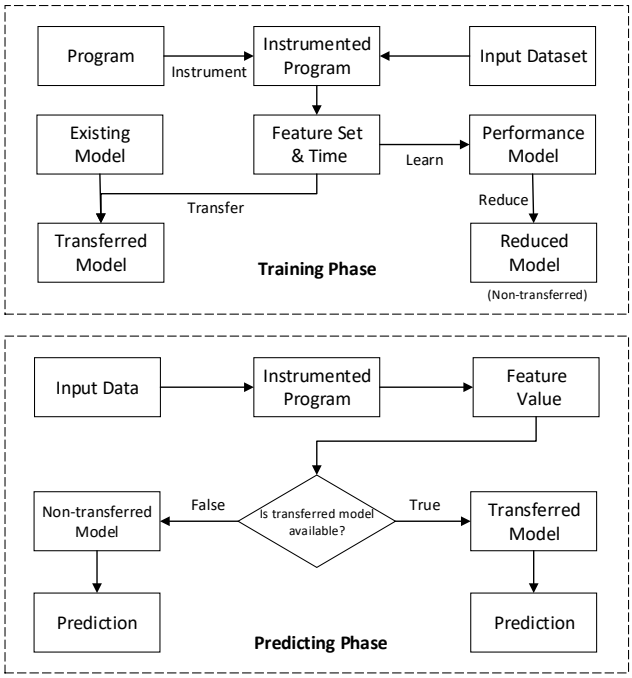


Fig. 1: Overview of automated performance modeling of HPC applications using machine learning.

The rest of this paper is organized as follows. Section 2 describes our method of modeling and predicting the performance in detail. Section 3 presents the experimental results and analyzes our method. Section 4 reviews a series of existing studies relevant to this research. Section 5 summarizes this study and discusses our plan for future studies.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2964767, IEEE Transactions on Computers

2 METHODOLOGY

2.1 Overview

Performance prediction is arguably a challenging problem. Theoretically, it is impossible to find a perfect prediction for every program (e.g. the halting problem). In practice, it is also difficult to predict the performance of a program driven by dynamic factors in its entire execution period (e.g. many randomized algorithms). In this work, we aim for modeling a program of which execution time mainly depends on its early execution phase. This research focus is inspired by the fact that a typical HPC application consists of three phases: initialization, repetitive calculation, and termination. Among these three phases, the initialization phase is usually used to define what will be calculated and how to be calculated [43].

Our method of performance modeling and prediction includes two phases: a training phase and a predicting phase, as shown in Fig 1. The training phase is used to collect data and build a performance model. It mainly consists of four stages: instrumentation, model learning, feature reduction, and model transfer. The predicting phase is used to handle a new input data of the target program, calculate the value of features, and output a predictive execution time with the transferred model or the non-transferred model depending on whether the transferred model is available. Next, we describe the processes of our method in detail.

2.2 Instrumentation

To capture behavior patterns of parallel programs without domain knowledge, we collect the runtime features through instrumentation. Instrumentation is a dynamic analysis for a program, which extracts program features from sample executions. We develop an instrumentor using clang [1]. The instrumentor automatically analyzes the abstract syntax tree (AST) of the source code of the target program, and inserts detective code around assignments, branches, loops, and communications to generate the instrumented program. Assignments reflect the data flow of a program. Branches and loops reflect the control flow. MPI communications can be regarded as the skeleton of a program [45]. To reduce the overhead of instrumentation, the inserted code keeps lightweight, like an incrementing integer counter for each branch feature and loop feature, and an assignment for each assignment feature [28]. We describe the instrumentation of these different types of features, respectively, below.

2.2.1 Assignments

The size of a problem and the amount of calculation are decided by key variables like the problem size, iteration count, convergence condition, and solution accuracy. To discover the key variables from the source code, we insert the instrumentation code after assignments. If a variable is assigned twice or more, all values are recorded as different features. We do not instrument the variables in a loop, because their values are updated frequently. Recording all versions of these variables is impractical. Code fragment 1 demonstrates an example of instrumentation for assignments.

```
1  //Code fragment 1
2  n=parse();
3  variable[1]=n;//instrument
4  n=preprocess(n);
5  variable[2]=n;//instrument
6  result=init();
7  variable[3]=result;//instrument
8  while(i<n)
9  {
10     result=calculate(i,result);
11     i=i+1;
12 }
```

2.2.2 Branches

Branches can lead to different execution paths, which may have significantly different execution times. Thus the results of conditional statements in branches are important features for predicting performance. This type of feature is difficult to fetch via domain knowledge or static code analysis. For example, code fragment 2 shows a common process that examines whether a file is successfully opened. If successful, the program will load data from the file and execute a heavy calculation, otherwise the program exits immediately. We cannot predict the result of this branch according to the file path string until the branch is actually executed. This example also indicates that black-box performance modeling that only considers program input parameters as features is insufficient, since some important features can often only be fetched at runtime. We insert instrumentation code after the conditional statement of branches and record their results as runtime features.

```
1  //Code fragment 2
2  fp=fopen(filename,'r');
3  if(fp)
4  {
5      if_counter[1]++;//instrument
6      load_data(fp);
7      calculate();
8  }
9  else
10 {
11     if_counter[2]++;//instrument
12     return;
13 }
```

2.2.3 Loops

Loops are usually the main calculation kernel of a program. The amount of iterations of loops has a direct impact and correlation with the performance. Code fragment 3 demonstrates an example of inserting instrumentations to count the number of iterations of loops, including nested loops. This type of instrumentation can introduce huge overhead, especially when the loop is deeply nested and the loop body is fine-grained. Section 2.4 will describe the reduce method, which can remove unnecessary features and corresponding instrumentations.

```
1  //Code fragment 3
2  while(i<n)
```

```
3  {
4      loop_counter[1]++;//instrument
5      for(int k=0;k<n;k++)
6      {
7          loop_counter[2]++;//instrument
8          calculation();
9      }
10     i=i+1;
11 }
```

2.2.4 MPI communications

Existing approaches can measure the communication characteristics of a program using benchmarks or communication traces. A benchmark can measure performance features of a network, like latency, bandwidth, or other parameters of models like LogP [17] and its variants [5], [27]. However, a benchmark mainly focuses on the general performance of a network, and it does not capture the details like data movement size, communication type, communication group of an MPI communication function call in an execution, etc. Using benchmarks to predict the communication time cost of a certain execution still needs an in-depth understanding of the application. Communication tracing can capture this information automatically, but tracing MPI events of a parallel program completely generates traces that often need very complex analysis, even if the program is a fixed-behavior benchmark program [45].

In our method, we do not aim for modeling the communication behaviors of a parallel program precisely. It meets our needs as long as we can characterize a statistical correlation between performance and a small number of communication features. We take the data size and the number of targets of MPI communication function calls to represent communication features. MPI communication functions, such as MPI_Send, MPI_Bcast, MPI_Gather, MPI_Allgather, MPI_Reduce, MPI_Allreduce, are instrumented before they are invoked. Code fragment 4 shows an example of instrumenting MPI communications. To minimize the overhead from synchronization, each MPI process maintains its local features during execution and synthesizes these features to the root process at the end of the program.

```
1  //Code fragment 4
2  //instrument
3  data_size[1]=n*sizeof(MPI_INT);
4  //instrument
5  comm_size[1]=MPI_Comm_size(my_comm);
6
7  MPI_Bcast(data,n,MPI_INT,root,my_comm);
8  calculate(data);
```

2.3 Model Learning

After collecting runtime features via instrumentation, we then try to discover the correlation between features and program execution time. It can be treated as a multivariate nonlinear regression problem. Assume that there are $\boldsymbol{n}$ samples. Each sample is expressed as (x, y), where x is a vector consisting of $\boldsymbol{m}$ features and y is the corresponding execution time. The goal of this regression problem is to find a mapping relation, $f : x \rightarrow y$, that minimizes the mean square error (MSE) between the predictive value and the real execution time in the n samples:

$$min\ MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2 \quad (1)$$

There exist numerous approaches to solve this regression problem, like ridge regression [11], LASSO [42], artificial neural network (ANN), and random forest [12]. We adopt a random forest approach with an optimization called extremely randomized trees [20]. Random forest is widely used in classification or regression tasks. Besides the capability of modeling complex nonlinear data, another advantage of random forest is that it can process mixed different types of features including float, integer, and enumeration [25]. Such a characteristic makes it suitable to model the runtime features we trace from MPI program execution. Besides, random forest can analyze the importance of each feature. It enables reducing redundant features and corresponding instrumentations.

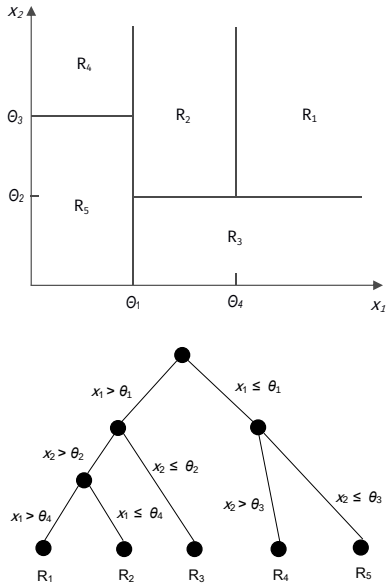


Fig. 2: A data space (top) consisting of two features (x_1, x_2) is divided into five regions (bottom)

Random forest is an ensemble learning method based on CART regression trees [13]. Ensemble learning trains many weak models that may have low prediction accuracy and synthesizes them to a strong model. A random forest consists of multiple regression trees. Figure 2 shows an example of a data space of two features, x_1 and x_2, and a regression tree divides the data space into five regions.

A node in a regression tree selects a feature value θ_i to divide the input data space into two disjoint regions. The regression tree recursively divides the input data space into multiple disjoint regions R_i. Each region is denoted by a leaf node and represents a response value, which means the program execution time in this study. When handling a new input, we can find a path from the tree root to a leaf node according to the value of features of this input, decide which region this input should belong to, and then take the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2964767, IEEE Transactions on Computers

response value as a prediction. This inference process also fits the nature of human decision. The model of a regression tree can be presented as follows:

$$f(x) = \sum_{i=1}^{k} (c_i I(x \in R_i)) \tag{2}$$

where k is the number of regions and c_i is the response value. In other words, c_i is the predictive execution time of samples in region R_i. $I(\cdot)$ is an indicator, which outputs 1 if x belongs to region R_i and 0 otherwise.

To achieve the optimization goal in Formula 1, c_i for leaf nodes and θ_i for inner nodes should be carefully considered. The division should ensure that each region contains similar data, where the similarity is measured by MSE between θ_i and all response values in this region. Generally, if a division feature is selected, it is easy to prove that the average of the corresponding y of x in region R_i can minimize the MSE [13]. The best division feature can be selected by enumerating all features in the subset and selecting the one with minimum MSE.

It often leads to over-fitting with building only one regression tree to divide regions and to predict the response value. Thus, a random forest needs numerous regression trees. Each regression tree fits a random subset in the original features and learns from different parts of training data. After building all regression trees, when handling a new input, the average output of all regression trees is taken as the final prediction of a random forest. The ensemble of these trees constructs a robust and well-generalized model. Extremely randomized tree adopts a modification that, in each division, the division feature and its division value are randomly selected. This extremely random division can further improve the accuracy of the ensemble model.

A useful improvement is building a model from a series of new features that are generated from a basis function, instead of the original features. The transformed model has almost the same form to the original model in Equation 2, but replaces x with $\Phi(x)$, where $\Phi(\cdot)$ is a basis function.

$$f(\Phi(x)) = \sum_{i=1}^{k} (c_i I(\Phi(x) \in R_i)) \tag{3}$$

To predict the program execution time, the d-order polynomial expansion function is a reasonable and effective basis function [24], [25], [28], which enumerates the combination of production of original features less than or equal to $\boldsymbol{d}$ orders. It transforms the original $\boldsymbol{m}$ features to $\binom{\boldsymbol{m+d}}{\boldsymbol{d}}$ new features. For example, let $\Phi(\cdot)$ be a 2-order polynomial expansion function. The transformation of a vector x with two features (a_1, a_2) is:

$$\Phi(x) = (1, a_1, a_2, a_1 a_2, a_1^2, a_2^2) \tag{4}$$

This transformation is inspired by the fact that the time complexity of an algorithm is usually a polynomial function of its parameters. Even if it contains other forms of functions like logarithmic function or trigonometric function, a polynomial function can still provide an accurate approximation, according to the Taylor series expansion. Experimental results also confirm that this transformation can improve the accuracy of predictions.

2.4 Features Reduction

A straightforward solution of using the collected features as discussed in Section 2.2 and the modeling technique discussed in Section 2.3 will generate a fairly complex model, because a parallel program may contain many assignments, branches, loops, and communications. The polynomial expansion will further increase the number of features. Redundant features take significant storage and introduce extra overhead. Even if we could afford such heavy storage and calculation, according to the Occam's razor, an unnecessarily complex model tends to be over-fitting, which has a poor generalization performance. Therefore, it is strongly desired to filtrate these features to generate a reduced model.

2.4.1 Reduction by time

Some features are too expensive to fetch their values when predicting the execution time of a new input. For example, if taking a variable that is generated at the end of the program as a feature, we need to completely execute the program to fetch the value of this feature. Such an expense makes the prediction much less useful.

This problem can be solved by setting a time threshold that only reserves the features below the threshold. In each execution of the training phase, the time cost for obtaining a feature can be recorded. Since the training dataset contains many executions with different time costs, we then calculate the ratio between the average time cost of a feature and the average time cost of program executions. Long-time executions occupy a larger portion for calculating averages since they are more sensitive to the expense. This ratio generally measures the expense of obtaining a feature. If we set the time threshold to be 5%, then features of which the time ratio is greater than 5% will be removed. Meanwhile, the corresponding instrumentation codes are also removed. After removing these instrumentations, the overhead can be significantly reduced.

When setting a high time threshold, the reduction is ineffective. In contrast, if setting a low time threshold, some features that are closely relevant with performance may be removed and the prediction accuracy would decrease. The trade-off between reduction effectiveness and prediction accuracy depends on the characteristics of the target application. As discussed in [43], in HPC applications, many actions that appear in the early execution phase can be effective predictors of performance, like loading and distributing data, parsing key parameters (e.g. scale, initial value, step size, etc.), evaluating and selecting solvers, early part of iterative calculation, etc. In our experiments, we evaluated different time thresholds and verified that a low time threshold can effectively reduce redundant features and still maintain desired prediction accuracy.

2.4.2 Reduction by importance

After reducing features by time threshold, we can further remove some features that have less impact on performance. At first, we need to define $importance$, a quantitative metric for measuring the impact of every feature. When we adopt random forest to build a performance model, intermediate results can be used to analyze the importance of features. As we have described in Section 2.3, a node in a regression

tree has its corresponding division feature, division value, and MSE. After dividing data by this node, data within a new region have higher similarity to each other, therefore the sum of MSE (weighted by data size of corresponding regions) of children nodes is less than that of this node. The extent of MSE reduction can represent the importance of the division feature [12].

Algorithm 1 Calculate Importance(forest)

```
1: forest.importances=zeros(m)
2: for each tree in forest do
3:    tree.importances=zeros(m)
4:    for each node in tree do
5:       left = node.leftchild
6:       right = node.rightchild
7:       tmp = (node.data_size × node.MSE -
8:       left.data_size × left.MSE -
9:       right.data_size × right.MSE)
10:      tree.importances[node.feature] += tmp
11:   end for
12:   tree.importances /= tree.root.data_size
13:   forest.importances += tree.importances
14: end for
15: forest.importances /= forest.n_trees
16: forest.importances /= sum(forest.importances)
```

Algorithm 1 presents the pseudocode of calculating feature importance using random forest. Assume that a random forest model is trained and MSE of each node is calculated. The forest contains n_trees trees and m features. Each tree node contains $data\_size$ data samples in its corresponding data region. The division feature of a tree node is denoted by $node.feature$. In line 1 and line 3, we initialize the importance as zero. The for loop starting from line 4 calculates the weighted MSE decrease of each tree node. The decreases from the same feature are accumulated as shown in line 10. The feature importance of the forest is the average value of the feature importance of its trees. Finally, the sum of the feature importance of the forest is normalized to 100%.

After calculating importance, we can sort features in the descending order, set a threshold, and reserve top features with accumulative importance, not greater than the threshold. This process reduces useless features. The effectiveness of reduction by importance is verified in our evaluation.

Traditional dimensionality reduction techniques, like principal component analysis (PCA) or singular value decomposition (SVD), can effectively reduce the number of features. However, these techniques are not suitable for our purpose. New features generated from PCA or SVD are linear combinations of original features, therefore we need to reserve all the instrumentation of original features to fetch their values and calculate the value of new features. It does not help reduce the overhead and storage demand from instrumentation.

Reduction by importance is also different from finding program hot spots via profiling, although they have some similar processes like instrumentation, recording runtime information, and analyzing performance data. Hot spots are program snippets that occupy a large proportion of the whole execution time and might be performance bottlenecks. Important features in our work denote runtime information that is statistically related to the execution time. For example, high performance sparse matrix-vector production algorithms [16], [30], [31] analyze the data pattern of the input matrix and vector and select different parameters before performing an actual calculation. This selection is obviously related to the execution time, but often it is not a hot spot since it is a quick pre-process for performance optimization.

2.5 Model Transferring

The method we have discussed so far requires the training data and the testing data from the same platform since a fundamental assumption of a typical machine learning method is that training data and testing data follow the same distribution. It largely limits the use scenarios of our performance model, since the platform settings may be modified or the application may need to be run on a completely new platform. It is feasible to repeat the modeling process, like collecting data, building model, and reducing features, on the new platform, but the new model may not achieve the desired accuracy until a sufficient amount of execution samples are collected and analyzed. This challenge is commonly referred as the cold start problem.

A possible solution to dealing with this problem is based on a critical observation that models on different platforms are not absolutely irrelevant to each other. The inherent behaviors of an application determine that it often maintains similar performance patterns across different platforms. A simple scenario is that two platforms have the same components except their CPU frequency has a slight difference. In this case, the performance model of one platform can be easily converted to that of another platform via arithmetic calculations. However, real-world scenarios are much more complicated, since modern computing systems consist of many components like CPU, memory, network, etc. Each of them is also a fairly complicated subsystem and is developed and updated rapidly. Therefore the relevance of performance models on different platforms is non-trivial. To leverage this relevance and to reuse an existing model, we adopt a transfer learning method.

In transfer learning, a $domain\ D = \{X, P(x)\}$ consists of a feature space X, where $x = (x_1, ..., x_m) \in X$, and a marginal probability distribution $P(x)$. A $task\ T = \{Y, f(\cdot)\}$ consists of a label space Y and an objective predictive function $f(\cdot)$. Note that the meanings of symbols remain the same as in section 2.3. Generally, our model transferring problem can be regarded as an inductive transfer learning problem [34]. Given a source domain D_s and its task T_s, a target domain D_t and its task T_t, we aim to find the target predictive function $f_t(\cdot)$ in T_t using the knowledge in D_s and T_s, where $D_s = D_t$ but $P(y_s|x_s) \neq P(y_t|x_t)$.

We introduce an instance-transfer method to solve this problem. An instance refers to an execution sample. An execution sample of an application on a certain platform can be regarded as a measure of this platform, and the measurement result is represented by a $(m+1)$-dimension point $(y, x_1, ..., x_m)$. Thus on two different platforms, multiple executions will generate two point sets with different marginal probability distributions of y in the same $(m+1)$-dimension

space. Although we do not know the exact distributions, we can learn a transfer function $h(\cdot)$ from the source platform to the target platform by:

$$min\ MSE = \frac{1}{u}\sum_{i=1}^{u}(y_{ti} - h(y_{si}, x_{ti1}, ..., x_{tim}))^2 \quad (5)$$

To solve this regression problem, it still requires that the target platform has executed a small number of execution samples $(y_{ti}, x_{ti1}, ..., x_{tim})$, where $i \in [1, u]$. Then we use the existing source model to generate execution time predictions as $y_{si} = f(\Phi(x_{ti1}, ..., x_{tim}))$, where $f(\Phi(x))$ is described in Equation 3. Another random forest model is applied to these samples to learn the transfer function $h(\cdot)$. It converts a prediction from the source model to a prediction from the target model. The final form of the target model is $h(f(\Phi(x)), x)$.

2.6 Predicting Phase

The predicting phase of our method is simpler than the training phase. There are two types of predictions: (1) using a non-transferred model to predict the performance on the $target$ platform with all trace data from this platform itself, and (2) using a transferred model to predict the performance on the $target$ platform with help from a $source$ platform.

Algorithm 2 Predict$(instru_app, inst, h, f_s, f_t, \Phi)$

```
x = instru_app(inst)
if h = NULL || f_s = NULL then
    return f_t(Φ(x))
else
    return h(f_s(Φ(x)), x)
end if
```

Algorithm 2 shows the procedure of predicting the execution time of an application with an input instance on the $target$ platform. Its parameters include the instrumented application $instru\_app$, an input instance $inst$ of the application, the transfer function $h(\cdot)$, the performance model on the source platform $f_s(\cdot)$ (a random forest trained by all trace data from the $source$ platform), the performance model on the target platform $f_s(\cdot)$ (a random forest trained by all trace data from the $target$ platform), and the polynomial expansion function $\Phi(\cdot)$. First, it executes the instrumented application with $inst$. During this execution, instrumentation codes export runtime features x. Then this execution will be terminated early according to the time threshold set by the feature reduction step.

If a source model or a transfer function does not exist, we can only make a prediction with the model $f_t(\cdot)$. As described in Section 2.3, each regression tree in the forest calculates its response according to feature values with polynomial expansion. Finally the model outputs the average response of all trees as the predicted execution time of a given input. If a source model $f_s(\cdot)$ and a corresponding transfer function $h(\cdot)$ exist, we first use $f_s(\cdot)$ to produce a prediction, and then transfer the prediction to the target platform. In general, both the non-transfer model $f_t(\cdot)$ and the transfer model $h(f_s(\Phi(x)), x)$ can predict the execution time of $inst$ on the $target$ platform. The transfer model can achieve better prediction accuracy if the training data set is small. When the amount of training data increases, the transfer model can still achieve similar accuracy like a non-transferred model. Thus, we always adopt the transferred model as long as it is available.

TABLE 1: Input parameters of Graph500.

Parameter	Type	Range
SCALE	integer	[10, 18]
EDGEFACTOR	integer	[10, 100]
N_PROC	integer	[16, 1024]

TABLE 2: Input parameters of GalaxSee.

Parameter	Type	Range
N	integer	[5000, 10000]
ROTATION_FACTOR	float	[0.1, 1.0]
SCALE	integer	[10, 1000]
MASS	float	[100.0, 100000.0]
INT_METHOD	enumeration	[1, 6]
FORCE_METHOD	enumeration	[1, 3]
N_PROC	integer	[16, 1024]

Besides making point prediction (e.g. scalar value of execution time), our model can also predict an interval of performance. It enables many useful functions, such as quantifying performance variation and detecting abnormal performance. The same code run with different background traffic can result in different performances. An interval prediction can provide more comprehensive insight into the varying performance.

The prediction of performance interval is based on the empirical variance of random forest. A leaf node in a regression tree, which represents a part of the feature space, may contain more than one training data. When making point prediction, the regression tree returns the mean value of the data in a leaf node as the prediction, while the variance can reflect the uncertainty of this prediction. Assume that each tree generates predictive mean μ_i and predictive variance σ_i^2. When a leaf only contains one data, we set a small number (e.g. $\sigma_{min}^2 = 0.01$) as its variance. The random forest with b trees can predict [25] joint mean μ and variance σ^2 across all trees by :

$$\mu = \frac{1}{b}\sum_{i=1}^{b}\mu_i \quad (6)$$

$$\sigma^2 = \left(\frac{1}{b}\sum_{i=1}^{b}\mu_i^2 + \sigma_i^2\right) - \mu \quad (7)$$

Then we can make interval predictions as $[\mu - \sigma, \mu + \sigma]$, which represents the uncertainty of performance predictions learned from training data.

3 Evaluation

In this section, we present the evaluation results and analyses of our method.

TABLE 3: Input parameters of SMG2000.

Parameter	Type	Range
nx,ny,nz	integer	[50, 200]
cx,cy,cz	float	[0.1, 10.0]
SOLVER	enumeration	[0, 3]
N_PROC	integer	[16, 1024]

3.1 Experimental Setup

3.1.1 Applications

Three applications, Graph500, GalaxSee, and SMG2000, were tested to predict their execution time under different input parameters.

Graph500 (version 2.1.4) [3] is a widely used benchmark focusing on data intensive computing. The main kernel of Graph500 is a Breadth-First Search (BFS) of a graph which starts with a single source vertex.

GalaxSee [2] is a parallel N-body simulation program used for simulating the movements of multiple celestial objects. It contains categorical parameters that determine different implementations of algorithms to solve the problem.

SMG2000 [4] is a parallel semicoarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation. This solver is a key component for achieving scalability in radiation diffusion simulations.

Among these three applications, Graph500 is a simple application since it contains few input parameters and each parameter has a straightforward impact on performance. In contrast, GalaxSee contains different types of parameters, and their impact on performance is not obvious. Table 1, 2, and 3 show their parameters and value range in our experiments, respectively.

3.1.2 Platforms and Environment

The experiments were conducted on three different platforms, denoted as A, B, and C. Table 4 lists the configuration of each platform. These three platforms have different characteristics. Platform B has higher single-core performance, but the number of cores per node is only 4. Platform C is a fat node with 8 low-frequency, 18-core CPUs. All communications in Platform C are intra-node communications. A single node of Platform A is in between Platform B and Platform C, but the size of the entire Platform A is much larger than that of B and C. Since the maximum number of CPU cores in platform B and C is 160 and 144, respectively, the parameter N_PROC, namely the number of processes, of each application is set to $[16, 128]$ on these two platforms.

Applications were compiled using Intel C/C++ compiler 15.0.0 and ran on CentOS 7.3 system. The regression model and the model transferring programs were written in Python 3.6.1 and scikit-learn library [35]. The Intel MPI version 5.0 was used as the MPI library.

3.2 Feature Reduction

We first evaluated the feature reduction methods introduced in Section 2.4. In this series of experiments, each application was tested 100 times on Platform A with different input parameters to trace runtime features. Since a $d-$order polynomial expansion on m features will generate $\binom{m+d}{d}$ new features, in this series of tests, we did not adopt polynomial expansion. We aim for reducing the number of features to under 20 so that a 3-order polynomial expansion will not generate too many new features.

The effectiveness of reductions is controlled by the time threshold and the importance threshold as defined in Section 2.4. We first evaluated the impact of varying time threshold settings. Table 5 shows the evaluation results, taking GalaxSee as an example. The actual time means the real time proportion of fetching feature values under its corresponding time threshold. For example, when setting the time threshold to be 5% of the complete program execution time, the actual time proportion we measured is 1.5%. The actual value is always smaller since the threshold is an upper bound. We took 50% of data as training set and used random forest to predict the others. The prediction error in our experiments is calculated by the below formula:

$$\frac{1}{n}\sum_{i=1}^{n}\frac{|y_{predict}[i]-y_{real}[i]|}{y_{real}[i]}\times 100\% \tag{8}$$

where $y_{predict}[i]$ is the predicted performance and $y_{real}[i]$ is the actual performance of the ith testing sample. Table 5 reports these results, which confirms our hypothesis discussed in Section 2.4 that lower time threshold can achieve better reduction, but it may reduce some useful features and decrease the prediction accuracy. When setting the time threshold to be 100%, it can achieve a very low error rate, but it is almost useless since the actual time and the overhead are impractical. The 5% time threshold is sufficient to achieve acceptable accuracy, and meanwhile to keep low actual time and overhead. Note that in this series of evaluation tests, we only ran 100 samples since some of the full instrumented programs took too much time. These samples were used to analyze the trend of the impact under varying thresholds. The errors measured in these tests are higher than those presented in Section 3.3 with the same reduction setting, since the latter was evaluated with more data samples.

We also evaluated with varying the importance threshold on GalaxSee with 5% time thresholds, and the results are shown in Table 6. With the decrease of the importance threshold, the number of reserved features also decreases, but the error rate becomes higher. The results of 95% threshold in Table 6 indicate that, after reduction by 5% time as shown in Table 5, there still exist many redundant features among 84 features. Removing these redundant features only slightly increased the error rate from 22.1% to 23.1%. Further

TABLE 4: Configuration of experimental platforms.

	Platform A	Platform B	Platform C
CPU type	E5-2680v4	E3-1240v5	E7-8860v4
frequency	2.4GHz	3.5GHz	2.2GHz
#cores/node	28	4	144
mem/node	128GB	32GB	1TB
#nodes	40	40	1
network	100Gbps OPA	100Gbps InfiniBand	N/A

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2964767, IEEE Transactions on Computers

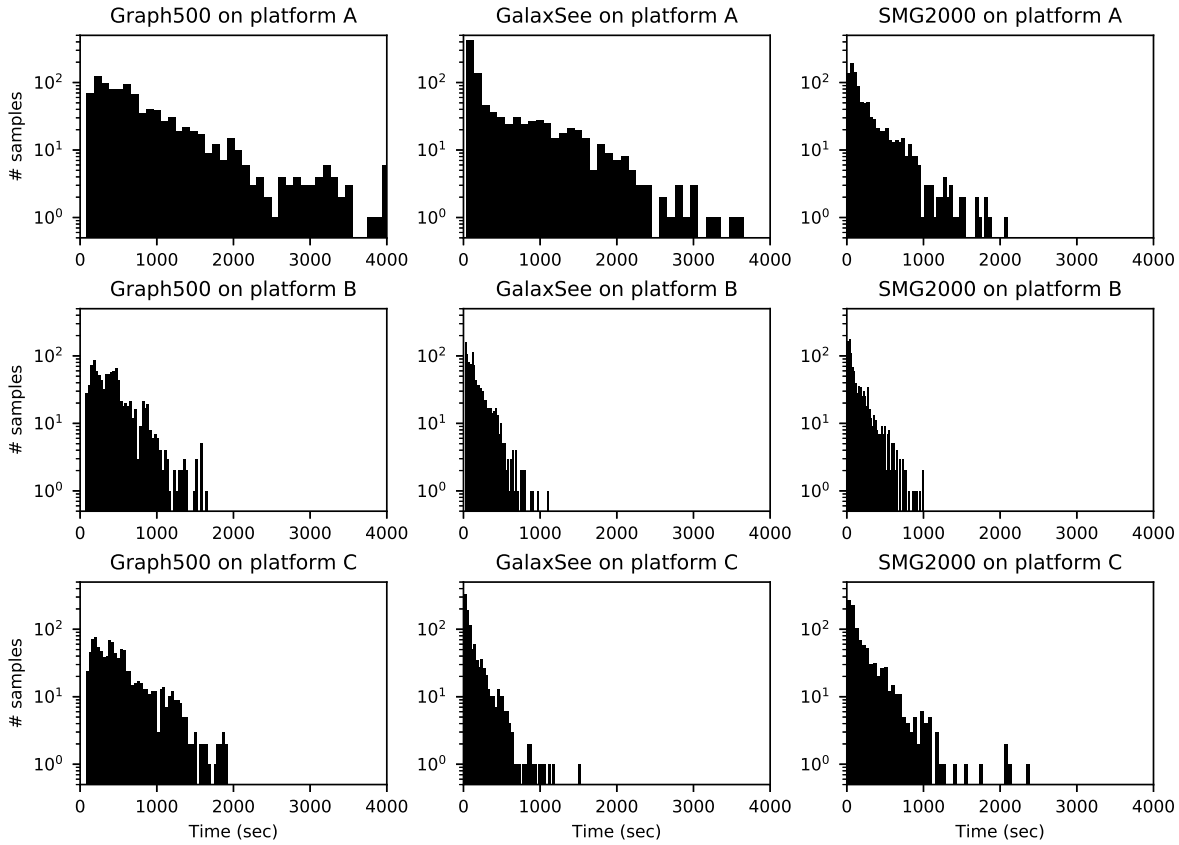


Fig. 3: Execution time distribution of three applications.

TABLE 5: The impact of different time thresholds on GalaxSee.

Time Threshold	Actual Time	# Features	Overhead	Error
5%	1.5%	84	1.2%	22.1%
10%	5.8%	85	1.2%	21.3%
25%	23.7%	124	29.4%	19.7%
50%	33.8%	126	29.6%	14.6%
75%	58.2%	132	32.8%	9.4%
100%	99.9%	357	1521.2%	7.5%

TABLE 6: The impact of different importance thresholds on GalaxSee.

Importance Threshold	# Features	Error
99%	29	22.7%
95%	18	23.1%
90%	15	27.7%
80%	8	33.6%
70%	3	45.3%

reduction may remove some important features and result in a higher error rate.

We then evaluated all three applications with 5% time threshold and 95% importance threshold. Table 7, 8, and 9 report these experimental results.

In general, feature reduction can effectively reduce redundant features as well as instrumentation overhead and storage. Taking GalasSee as an example, and as Table 8 shows, the complete number of instrumentations of GalaxSee is 357. These instrumentations introduced a total of extra 1521.2% overhead, which means that running the instrumented GalaxSee costs more than ten times of execution time than the original program without any instrumentation. In each run, the trace data generated from instrumentation required 5,164KB storage on average. After two processes of reduction, only 18 important features were reserved. The storage demand was reduced to 51KB. Their instrumentations only introduced 0.5% overhead, which means that running an instrumented program is almost the same as the original one. We can provide the instrumented version of programs for HPC users to run their jobs. It will continuously generate runtime feature data and help the regression model to be more accurate.

TABLE 7: The impact of feature reduction on Graph500.

Feature state	# Features	Overhead	Storage (per run)
full	87	71.6%	1,458KB
by time	15	1.4%	85KB
by importance	5	1.1 %	28KB

TABLE 8: The impact of feature reduction on GalaxSee.

Feature State	# Features	Overhead	Storage (per run)
full	357	1521.2%	5,164KB
by time	84	1.2%	615KB
by importance	18	0.5 %	51KB

TABLE 9: The impact of feature reduction on SMG2000.

Feature State	# Features	Overhead	Storage (per run)
full	648	29.4%	7,821KB
by time	87	0.5%	697KB
by importance	17	0.1 %	151KB

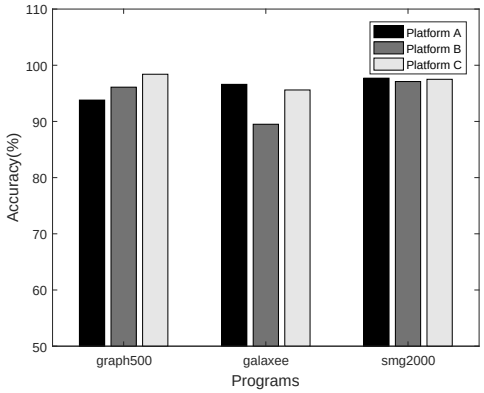


Fig. 4: The prediction accuracy of performance rank.

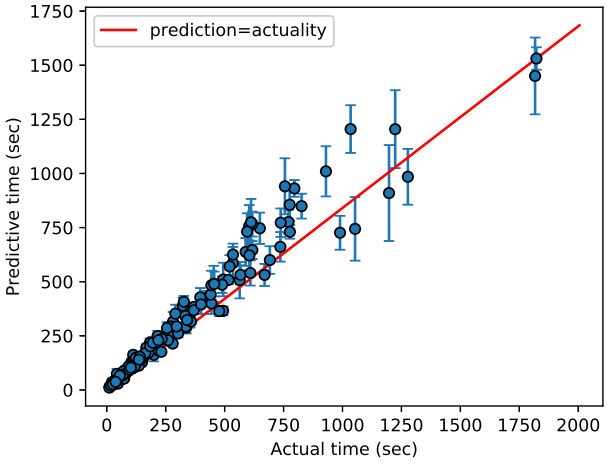


Fig. 5: Predicting performance interval of SMG2000.

3.3 Performance Prediction

In this section, we discuss the prediction accuracy of different machine learning methods for our performance model.

We ran each application on each platform 1,000 times with different input parameters. In other words, each modeling task had 1,000 data samples. The input parameters we used for experiments were generated from the parameter value range of each application uniformly and randomly. Figure 3 shows the time distribution of three applications.

After fetching runtime feature values of these samples, each feature was normalized to zero mean and unit variance independently. Part of data samples was randomly selected as the training set while others were used as the testing set.

Figure 6 presents the mean errors of the testing set under different ratios of the training data. The regression methods we tested include least absolute shrinkage and selection operator (LASSO), support vector machine regression (SVR) with radial basis function (rbf) kernel, and random forest (RF). Each method was applied to both the raw form of data and its 3-order polynomial expansion. We also tested two sophisticated modeling methods from related works. One is a linear model based on performance model normal form (PMNF) [9], [10], [14]. The other one is a deep learning model called PerfNet [32]. PMNF contains a special form of nonlinear expansion. PerfNet does not have a process for feature reduction so that the polynomial expansion would make the model unnecessarily large and possibly overfitting. Therefore we did not use polynomial expansion on PMNF and PerfNet.

Since the mapping relation between the features and the execution time is complicated, a linear regression method, like LASSO, has higher prediction error than nonlinear methods. Using polynomial basis function, it is actually converted to nonlinear methods, and the corresponding errors are significantly reduced. It indicates that polynomial function is an acceptable approximation between features and program execution time.

As of nonlinear regression methods, SVR, PerfNet and RF can achieve lower prediction error. Polynomial expansion provides little benefit to SVR and RF in most cases as shown in Figure 6. Generally, RF has better prediction accuracy. For a simple program like Graph500, PerfNet can achieve comparable results. An important factor of the prediction accuracy is the impact of categorical features. A categorical feature is a variable of which value belongs to a finite and discrete set. The value of a categorical feature is just a tag and does not have a numerical meaning. Thus, if a regression model considers it as a numerical feature, the result can be worse. Although there are many transformation measures to avoid this problem, they need a precondition to verify which one is a categorical feature. However, because we do not have domain knowledge, we cannot realize which feature is categorical and adopt transformation measures for it. The advantage of RF is that categorical features naturally have less impact on it. A regression tree in RF can generate nodes to divide the sample data by a categorical feature, then within each divided data region, the categorical feature is a constant and has no impact on further regression. In our experiments, GalaxSee contains categorical features, therefore the superiority of RF is more apparent than that in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2964767, IEEE Transactions on Computers

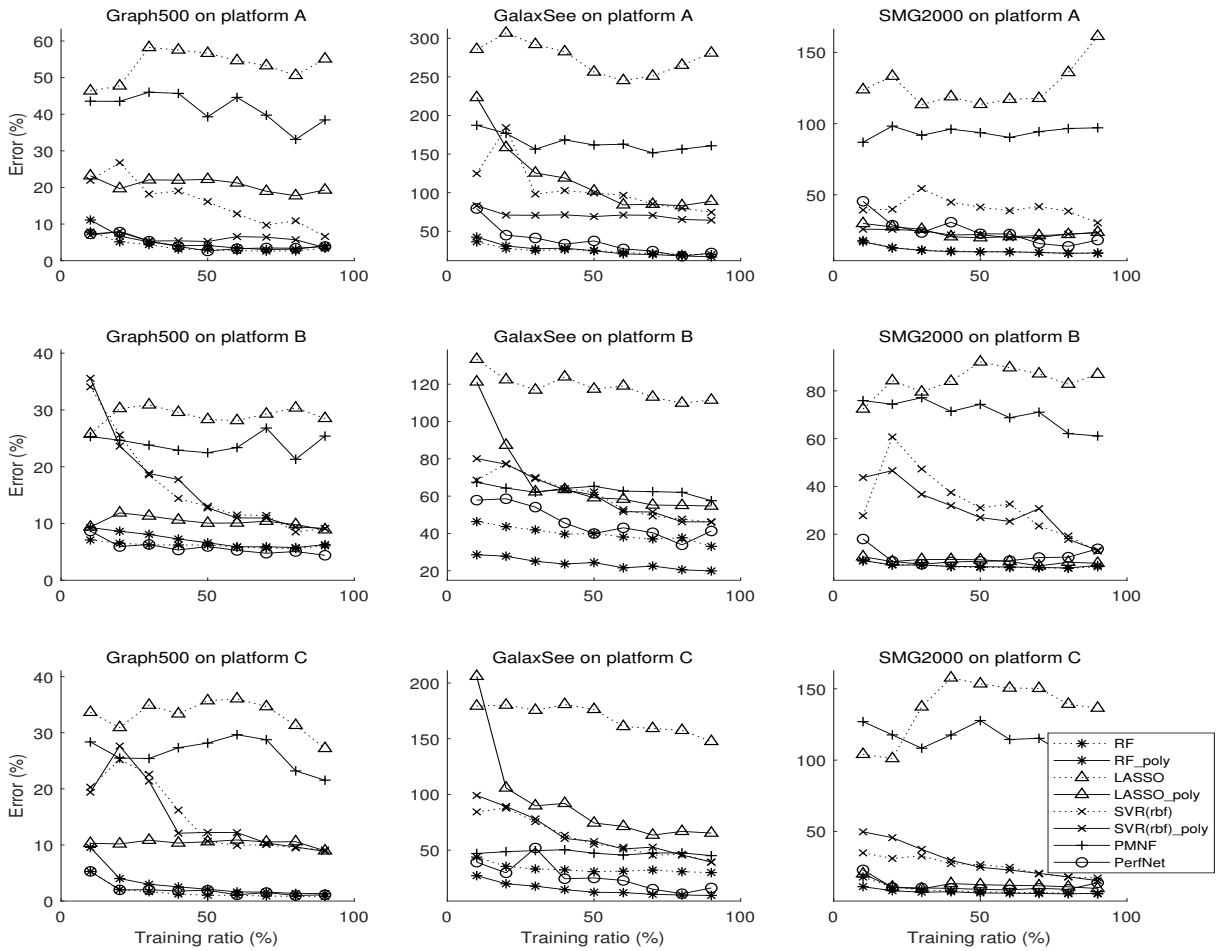


Fig. 6: The prediction error of different machine learning methods with various applications and platforms.

experiments with other applications.

In addition to the execution time prediction, another objective that attracts much attention in many applications is the relative rank of performance on different input parameters. Sometimes we just want to understand that which input will be faster while the exact execution time is not necessary [15], [23], [32]. To measure the ability of predicting the performance rank using our model, we took 50% data as the training set and tested the accuracy of predicting the performance rank of all sample pairs in the testing set. Assume that the testing set contains k samples, the accuracy is calculated as:

$$accuracy = \frac{2\sum_{i=1}^{k}\sum_{j=i+1}^{k}(z_{ij})}{k(k-1)} \times 100\% \tag{9}$$

where

$$z_{ij} = \begin{cases} 1 \ (y_{predict}[i] - y_{predict}[j])(y_{real}[i] - y_{real}[j]) > 0 \\ 0 \ otherwise \end{cases} \tag{10}$$

This objective measures the ratio of successfully predicting the faster input parameter in each sample pair where there are $\frac{k(k-1)}{2}$ pairs in total. Figure 4 shows the experimental results of three applications on three platforms. Considering all experimental configurations, the accuracy of predicting the performance rank of each sample pair is around 90%. It confirms that in most cases, our model can predict the correct rank relationship when comparing the performance of two different input parameters.

3.4 Performance variation

As we discussed in Section 2.6, besides predicting the scalar value of execution time, our model can also predict an interval of performance, based on calculating empirical variance. Figure 5 shows an example of predicting performance interval of SMG2000 on Platform A. We draw an interval $[\mu - \sigma, \mu + \sigma]$ for each prediction, as Equation 6 and 7 describe. Some points diverge from the ideal predicting line, thus they have large variance intervals. It indicates that these predictions have relatively high uncertainty and more training data around these points are required to improve their predictions.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2964767, IEEE Transactions on Computers

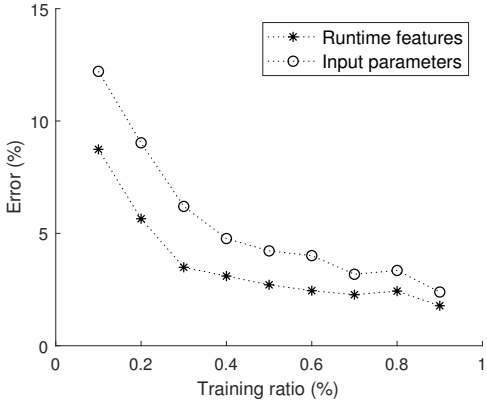


Fig. 7: Impact of runtime features on Graph500.

TABLE 10: Feature importance of Graph500.

No.	Importance	Type	Location
1	0.353	Variable	main.c line 75
2	0.188	Variable	main.c line 143
3	0.184	Variable	utils.c line 25
4	0.148	Variable	main.c line 76
5	0.125	Loop counter	utils.c line 32

3.5 Impact of Runtime Features

In this section, we discuss the impact of runtime features. Taking Graph500 on Platform A as an instance, we show the importance of each feature in Table 10 with descending order. The importance is defined and calculated by Algorithm 1. Since Graph500 does not require complicated domain knowledge, we can check the meaning of each feature from the source code. To clarify the meaning of features, this measurement of importance is conducted before the polynomial transformation. In Table 10, these features represent the problem scale, the topology of MPI communication, the number of processes, the average node degree and a for-loop counter, respectively. The value of the for-loop counter is logarithm of number of processes if it is required to be a power of two, otherwise zero. All these five features are automatically detected by our model. Among them, the first, the third and the forth features are directly covered by input parameters while the others are not. The three input parameters contribute about 0.685 importance in sum, which means they are quite important but not dominant. Further comparison is showed in Figure 7. With the same random forest model, if we only use input parameters as features, the predictive error will increase.

3.6 Effectiveness of Model Transferring

In this section, we discuss the effectiveness of model transferring. Similar to these tests reported in Section 3.3, there are 1,000 random data samples of each application available on each platform. The difference in the experiment setting is that we only take a small number of samples (1%-10%) as the training set and others (99%-90%) as the testing set.

The transfer method used an existing well-trained random forest model from another platform, discussed in Section 3.3, as a source model, and transferred it to a target model. We used Platform B, C as the source platform to generate the transferred model on Platform A. Figure 8 reports mean errors on the testing set under different ratios of the training data. As a comparison, a traditional machine learning method alone, like random forest, has worse results on predicting performance, because the training data is too small to learn a reliable model from it. With the increase of training percentage, the effectiveness of these two methods tends to be closer.

4 Related Work

In this section, we review and discuss existing studies along four categories, analytical modeling methods, replay-based modeling methods, statistical modeling, and model transferring for the performance prediction of parallel programs.

4.1 Analytical Modeling

Analytical modeling uses an analytical formula to describe the program performance. As we have introduced in Section 1, an analytical model is tightly coupled with a particular algorithm and a particular application domain, thus it involves extensive efforts from human experts. For example, Eller et. al. [18] proposed an analytical model for Krylov Solver on structured grid problems. Hang et. al. [36] proposed a detailed performance model for deep neural networks. Barker's model [8] focused on the Krak Hydrodynamics Application. In general, an analytical model for a specific application is difficult to be applied to other applications.

4.2 Replay-based Modeling

Replay-based modeling uses instrumentation or similar techniques to trace detailed information from program executions. Through analyzing the trace, a synthetic program can be reconstructed for replaying behaviors of the original program and predicting its performance. Since tracing and analysis can be automated, this type of modeling and prediction method can eliminate the requirement of domain knowledge and can be generalized for different applications.

Several studies exist in this area. For instance, Sodhi, Zhang and Hao [21], [38], [45] constructed a skeleton of a parallel program from traces. Skeleton preserves the flow and logic of the original program but reduces calculations and communications. Zhai et. al. [44] analyzed traces and introduced a deterministic replay to predict the performance. However, traces require a large storage space. Even when tracing simple parallel programs like NPB benchmark [7], storage requirements can range from hundreds of megabytes to tens of gigabytes for each run [9], [44]. Besides, a trace-based modeling usually consists of a program skeleton or other forms of a synthetic program. A synthetic program shrinks computation and communication of the original code. It loses the semantic of the original code; therefore, it is not human-readable. The prediction lacks interpretability, which does not locate the performance factors of parallel programs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.2964767, IEEE Transactions on Computers

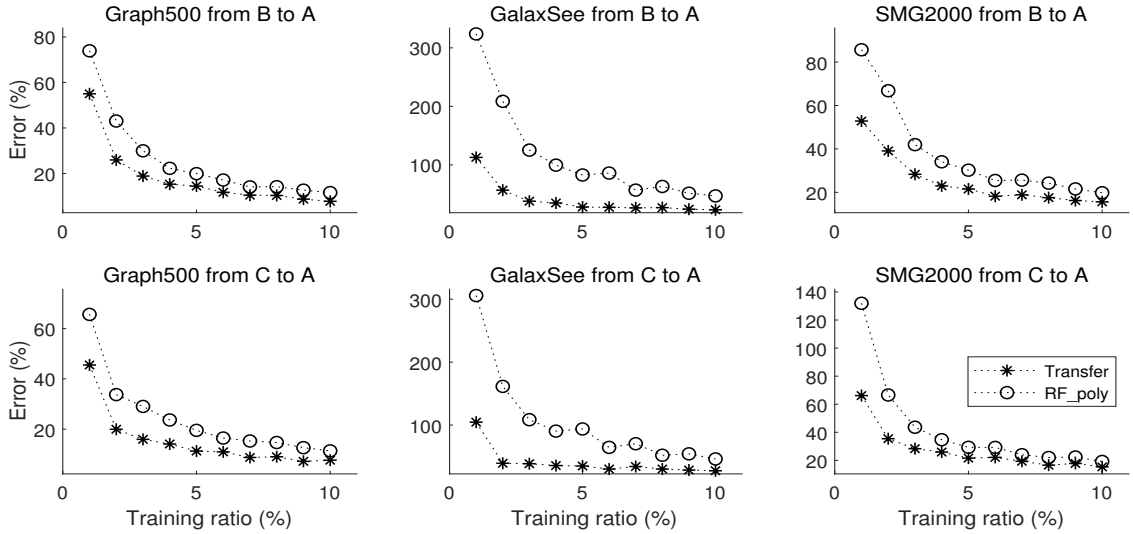


Fig. 8: The prediction error of transfer method and random forest with various applications and platforms. Transfer method uses an existing model from a source platform to help build the model on the target platform, while random forest method can only exploit data from the target platform.

4.3 Statistical Modeling

The development of machine learning techniques enables the possibility of empirically analyzing the performance patterns of a parallel program under different input parameters. Song et. al. [39] adopted a machine learning method called Delta Latent Dirichlet Allocation (ΔLDA) [6] to model application executions. It can locate possibly low-performance code blocks but not predict the exact execution time cost. Ogilvie and Thiagarajan [33], [41] focused on constructing surrogate models for auto-tuning. In these problem settings, performance models are mainly required to achieve good accuracy on high-performance sub-space while low-performance part can be ignored. Lee et. al. [29] proposed methods that employ artificial neural networks to predict the performance of parallel programs. Their methods can capture system complexity implicitly from various input data, but their work only focuses on a fixed number of cores. Additionally, their method cannot analyze the impact of each feature since the artificial neural network is a black-box model. A series of studies [9], [10], [14] attempted to model the performance of kernels in a parallel program with using linear regression methods like ridge regression, least absolute shrinkage and selection operator (LASSO), or their variants to model the relationship between features and execution time. Linear regression methods are easy to be implemented and their prediction results are concise and interpretable. However, since parallel programs can have complex behavior patterns, a linear model may not be accurate to characterize the performance under different input parameters. EPMNF transformation [9], [10] can be used to improve the nonlinear fitness of linear regressions, but before the transformation, domain experts are required to determine a small range of feature candidates.

4.4 Model Transferring

The traces of an application from a certain platform cannot be used directly when modeling the performance of the application on another platform. It is feasible to collect data and build another model on the new platform; however, it is troublesome and often unnecessary. A better solution would be transferring the existing model to reuse it or assist in building the new model. Numerous studies have been conducted, but these existing studies usually do not focus on the execution time prediction but the prediction of the performance rank of an application under different conditions, since transferring the rank correlation across different platforms is easier than transferring the execution time model. Hoste et. al. [23] used benchmarks to measure different platforms, then according to the similarity between benchmarks and the application of interest, their work can predict the performance rank of an application on different platforms. Chen et. al. [15] used the Bayesian network to capture the parameter dependencies of an application. Marathe et. al. [32] built a performance model with deep neural networks. They transferred neural networks with a fine-tuning method, which took a trained network for a platform to initialize the connection weights of a network for a new platform. Their work showed that deep neural networks do not outperform traditional machine learning methods (e.g. random forest) on execution time prediction in general, but they can help users find better application configurations on different platforms.

4.5 Comparison of This Study and Existing Studies

As a comparison of this research and existing studies, our method is a statistical modeling method and uses machine learning techniques to analyze historical executions and to predict the execution time of a parallel program. There are several critical differences between our work and existing work though. First, existing studies using machine learning techniques mainly consider input parameters as features to model the performance of parallel programs, whereas our work considers runtime features about variable assignments, branches, loops, and communications, which are

a superset of input parameters. Runtime features contain more information that may not be covered by input parameters, thus our method can achieve equal or higher accuracy for predicting performance, compared with models based on input parameters. On the other hand, there is sufficient difference between our method and existing works that also take runtime features into account. To effectively utilize runtime features and eliminate the negative effect of redundant features, our method automatically identify the important subset of all possible runtime features using time and importance reductions. Besides making scalar prediction of performance, our method can also predict performance interval, which enables quantifying performance variation and checking the confidence of a prediction. Moreover, our method can transfer an existing random forest-based performance model to a new platform with few historical program executions. It effectively alleviate the cold start problem, which is a common problem for statistical modeling.

5 Conclusion

In this paper, we introduce a novel method to model and predict the performance of parallel programs (MPI programs). We develop a tool to automatically analyze the syntax tree of MPI programs and instrument them, so that we can detect its runtime features related to computation and communication, without requiring any domain knowledge. We design a strategy to automatically analyze and fit the runtime feature data of MPI programs using random forest technique, thereby we can predict the performance. Since we adopt a lightweight instrumentation and further reduce features by two reduction processes, the overhead of instrumentation is low, with much less storage demand compared to existing methods. Combined with model transferring method, an existing performance model can be reused to predict the performance on a new platform with a small number of training samples.

Although we have achieved desired prediction on three tested applications that well represent typical HPC applications, the ability of our method can be further optimized. Since we only extract features from early execution phase of an application, if its behavior is not only decided by the early phase, our method may not have an accurate prediction. In the future, we will further investigate how to extract behavior patterns throughout the entire execution period of an application and further optimize our model.

ACKNOWLEDGMENTS

This study is supported by NSF of China (grant number: 61772485, 61432016). Experiments in this study were conducted on the supercomputer system in the Supercomputing Center of University of Science and Technology of China.

References

[1] Clang: a c language family frontend for llvm. http://clang._ llvm.org/.

[2] Galaxsee hpc module 1: The n-body problem, serial and parallel simulation. http://shodor.org/petascale/_ materials/UPModules/NBody/.

[3] Graph 500 reference implementations. http://www.graph_ 500.org/referencecode.

[4] The smg2000 benchmark code. https://asc.llnl.gov/computing_ resources/purple/archive/benchmarks/smg/.

[5] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. Loggp: Incorporating long messages into the logp model for parallel computation. *Journal of parallel and distributed computing*, 44(1):71–79, 1997.

[6] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In J. N. Kok, J. Koronacki, R. L. d. Mantaras, S. Matwin, D. Mladenič, and A. Skowron, editors, *Machine Learning: ECML 2007*, pages 6–17, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[8] K. J. Barker, S. Pakin, and D. J. Kerbyson. A performance model of the krak hydrodynamics application. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 245–254, Aug 2006.

[9] A. Bhattacharyya and T. Hoefler. Pemogen: Automatic adaptive performance modeling during program runtime. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 393–404. ACM, 2014.

[10] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler. Using compiler techniques to improve automatic performance modeling. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 468–479. IEEE, 2015.

[11] C. M. Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.

[12] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[13] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[14] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. Fast multi-parameter performance modeling. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 172–181. IEEE, 2016.

[15] H. Chen, W. Zhang, and G. Jiang. Experience transfer for the configuration tuning in large-scale computing systems. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):388–401, March 2011.

[16] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *SIGPLAN Not.*, 45(5):115–126, Jan. 2010.

[17] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993.

[18] P. R. Eller, T. Hoefler, and W. Gropp. Using performance models to understand scalable krylov solver performance at scale for structured grid problems. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 138–149, New York, NY, USA, 2019. ACM.

[19] E. Gaussier, D. Glesser, V. Reis, and D. Trystram. Improving backfilling by using machine learning to predict running times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 64. ACM, 2015.

[20] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[21] M. Hao, W. Zhang, Y. Zhang, M. Snir, and L. T. Yang. Automatic generation of benchmarks for i/o-intensive parallel applications. *Journal of Parallel and Distributed Computing*, 124:1–13, 2019.

[22] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, page 6. ACM, 2011.

[23] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere. Performance prediction based on inherent program similarity. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 114–122, Sept 2006.

[24] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems*, pages 883–891, 2010.

[25] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[26] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–37. ACM, 2001.

[27] T. Kielmann, H. E. Bal, and K. Verstoep. Fast measurement of logp parameters for message passing platforms. In *International Parallel and Distributed Processing Symposium*, pages 1176–1183. Springer, 2000.

[28] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: automatic performance prediction for smartphone applications. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, pages 297–308. USENIX Association, 2013.

[29] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. 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*, pages 249–258. ACM, 2007.

[30] J. Li, G. Tan, M. Chen, and N. Sun. Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication. *SIGPLAN Not.*, 48(6):117–126, June 2013.

[31] K. Li, W. Yang, and K. Li. Performance analysis and optimization for spmv on gpu using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):196–205, Jan 2015.

[32] 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*, SC '17, pages 31:1–31:12, New York, NY, USA, 2017. ACM.

[33] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. Minimizing the cost of iterative compilation with active learning. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 245–256. IEEE Press, 2017.

[34] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, Oct 2010.

[35] 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*, 12:2825–2830, 2011.

[36] H. Qi, E. R. Sparks, and A. Talwalkar. Paleo: A performance model for deep neural networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[37] H. Sanjay and S. Vadhiyar. Performance modeling of parallel applications for grid scheduling. *Journal of Parallel and Distributed Computing*, 68(8):1135–1145, 2008.

[38] S. Sodhi, J. Subhlok, and Q. Xu. Performance prediction with skeletons. *Cluster Computing*, 11(2):151–165, 2008.

[39] L. Song and S. Lu. Statistical debugging for real-world performance problems. *SIGPLAN Not.*, 49(10):561–578, Oct. 2014.

[40] D. Sundaram-Stukel and M. K. Vernon. Predictive analysis of a wavefront application using loggp. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 141–150, New York, NY, USA, 1999. ACM.

[41] 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*, pages 385–395. ACM, 2018.

[42] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[43] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 40–40. IEEE, 2005.

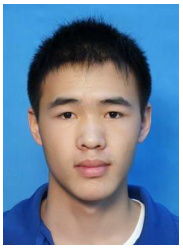
[44] J. Zhai, W. Chen, W. Zheng, and K. Li. Performance prediction for large-scale parallel applications using representative replay. *IEEE Transactions on Computers*, 65(7):2184–2198, 2016.

[45] W. Zhang, A. M. Cheng, and J. Subhlok. Dwarfcode: A performance prediction tool for parallel applications. *IEEE Transactions on Computers*, 65(2):495–507, 2016.

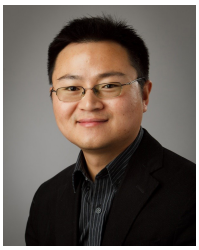
Jingwei Sun is a doctoral student at the University of Science and Technology of China. His research interests include distributed deep learning, high performance computing and algorithm optimizations.

Guangzhong Sun is an Associate Professor in School of Computer Science and Technology, University of Science and Technology of China. He is also a member of National High Performance Computing Center (Hefei) and the Head of Algorithm and Data Application (Ada) Research Group. His research focuses on high performance computing, algorithm optimizations, and data processing.

Shiyan Zhan is a graduate student at the University of Science and Technology of China. His research focuses on high performance computing.



Jiepeng Zhang is a graduate student at the University of Science and Technology of China. His research focuses on high performance computing.



Yong Chen is an Associate Professor and Director of the Data-Intensive Scalable Computing Laboratory in the Computer Science Department of Texas Tech University. He is also a Site Director of the Cloud and Autonomic Computing center at Texas Tech University. His research interests include data-intensive computing, parallel and distributed computing, high-performance computing, and cloud computing.