2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)

Automated Performance Modeling Based on Runtime Feature Detection and Machine Learning

Jingwei Sun[1], Shiyan Zhan[1], Guagnzhong Sun[1], Yong Chen[2]
[1]School of Computer Science and Technology, University of Science and Technology of China, Hefei, China
[2]Department of Computer Science, Texas Tech University, Lubbock, Texas, U.S.A.
{sunjw, zsynew}@mail.ustc.edu.cn, gzsun@ustc.edu.cn, yong.chen@ttu.edu

***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, assisting resource requirement estimation, etc. The performance of parallel programs is affected by numerous factors, including but not limited to hardware, system software, 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 scale, 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 with a new input. Our experiments and analyses of three parallel programs, Graph500, GalaxSee and SMG2000, on three different systems show that our method performs well, with less than 20% error in predictions on average.**

I. 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 and prediction of 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 [24], [13]. Therefore, the utilization of the entire HPC system can be improved. **Second**, the model can offer insights about the application behaviors [17], [15], 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 [29], [30]. According to the predicted performance, users can synthetically consider predictive computation time and expense, and then apply reasonable number of CPU cores from HPC system.

Building an accurate performance model of parallel programs 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, system software, applications, algorithms and input parameters. It is especially difficult to build a general-purpose model which synthesizes all aspects of factors.

In this paper, we focus on predicting the execution time of parallel programs, particularly MPI programs as MPI is the de facto standard parallel programming model, on an HPC cluster with different inputs and at scale. Previous studies mainly introduced two types of methods: *analytical modeling* [19], [27], [6] and *trace-based modeling* [26], [31], [30], [5], [7], [30]. An *analytical modeling* method has an arithmetic formula describing a parallel program performance and can make a prediction of execution time quickly. For example, the time complexity analysis of a program can be considered a very rough analytical model. 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 *trace-based model* is built from traces that are generated by instrumenting a program source code (or at binary code level without modifying the source codes) and running the modified program. Traces can capture useful information about computation, communication, and other runtime features of a parallel program. Through analyzing traces, a trace-based model can predict the execution time empirically without requiring domain knowledge and human efforts. However, the trace-based modeling usually requires large storage space to keep traces (ranging from hundreds of megabytes to tens of gigabytes for each run [7], [30]), and the instrumentation introduces extra overhead and slows down the application. Moreover, since trace-based model is constructed by program skeleton [31], [26] or other forms of synthetical program [30], human is difficult to understand the model and cannot extract performance features via it.

In this paper, we propose a performance modeling and prediction method that has low overhead and storage demand, and can extract important features from traces. It automatically inserts instrumentation code, detects runtime features, analyzes feature data, and uses a regression machine learning method to predict execution time and reduce features, therefore it is free from requiring domain knowledge and human efforts. We also

0-7695-6329-5/17/31.00 ©2017 IEEE
DOI 10.1109/ISPA/IUCC.2017.00115

adopt a lightweight instrumentation to reduce the overhead. A random forest regression approach[14] is used to model the execution time of parallel programs from varying input parameters and at different scales. This approach is also adopted to extract a subset and important features from original trace data to further reduce the number of instrumentations and the storage demand.

The contributions of this paper are summarized as follows:

- We propose a method to empirically 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 error of prediction 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.

The rest of this paper is organized as follows. Section II reviews a series of existing studies relevant to this study. Section III describes our method of modeling and predicting the performance in details. Section IV presents the experimental results and analyzes our method. Section V summarizes this study and discusses our plan of future studies.

II. Related Work

We review and discuss existing studies along three categories, analytical modeling methods, trace-based modeling methods, and trace-based modeling with machine learning methods for the performance prediction of parallel programs.

A. Analytical Modeling

Analytical modeling uses an analytical formula to describe the program performance. As we have introduced in Section I, 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, Kerbyson et. al. [19] proposed an analytical model for an application called SAGE (SAIC's Adaptive Grid Eulerian hydrocode). Sundaram-Stukel et. al. [27] proposed an analytical model for a complex wavefront application. Barker's model [6] focuses on the Krak Hydrodynamics Application. In general, an analytical model for a specific application is difficult to be applied to other applications.

B. Trace-based Modeling

Trace-based modeling uses instrumentation or similar techniques to trace detailed information from program executions. Through analyzing the trace, the program execution time can be estimated. Tracing and analysis can be automated, therefore 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 space. For instance, Sodhi and Zhang [26], [31] 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. [30] analyzed traces and introduced a deterministic replay to partially replay and predict the performance. However, traces require large storage space. Even when tracing simple parallel programs like NPB benchmark [5], storage requirement can range from hundreds of megabytes to tens of gigabytes for each run [7], [30]. In addition, a trace-based modeling usually consists of program skeleton or other forms of synthetic program. A synthetic program shrinks computation and communication of the original code. It loses the semantic of original code, therefore it is not human-readable. The prediction lacks of interpretability, which does not locate the performance factors of parallel program.

C. Trace-based Modeling with Machine Learning

Traditional trace-based modeling methods build model by tracing executions under fixed input parameters, because different input parameters correspond different program skeletons or replay behaviors. The development of machine learning techniques enables the possibility of analyzing the performance patterns of a parallel program under different input parameters. Ipek and Mckee [18], [21], [25] proposed a method that employs artificial neural networks to predict the performance of parallel programs. Their method 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. A series of studies attempted to [7], [12], [8] 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, linear model may not be accurate to characterize the performance under different input parameters.

D. Comparison of This Study and Existing Studies

As a comparison of this research and existing studies, our method is a trace-based modeling method and uses machine learning techniques to analyze traces and predicts the execution time of a parallel program. There are several critical differences between our work and existing work though. First, our method reduces the overhead from instrumentation with a lightweight and filtrated instrumentation, therefore the instrumented parallel program can normally execute and generate trace data meanwhile. On the other hand, existing studies using machine learning techniques mainly consider input parameters as features to model the performance of parallel programs,

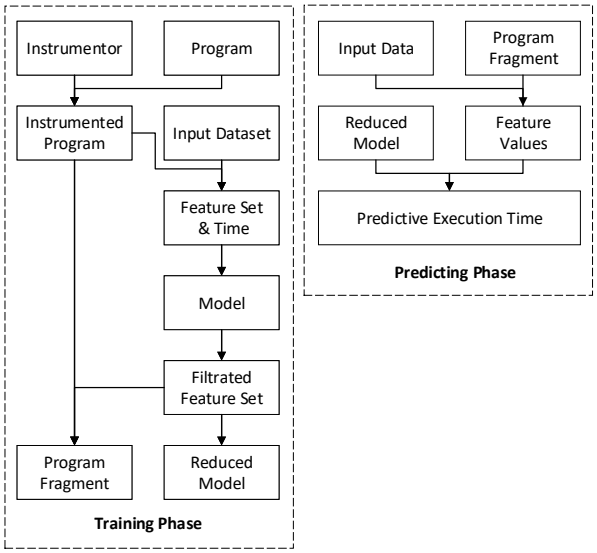

Fig. 1. Overview of our method for performance modeling.

whereas our work considers runtime feature about variable assignments, branches, loops and communications, which is a superset of input parameters, therefore our method captures much more comprehensive performance factors beyond input parameters. Our method also adopts an extremely randomized tree [14] to model the complex relationship between features and execution time. It achieves much higher accuracy than linear regression and can well interpret the importance of each feature.

III. Methodology

A. Overview

Our method of performance modeling and predicting 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 three processes: instrumentation, model building, and feature filtration. 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 using the trained model. Next we describe the processes of our method in detail.

B. 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 [31]. 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 [20]. In the following, we describe the instrumentation of these different types of features, respectively.

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 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. In addition, these variables are often used for temporary operations or as intermediate results of iterative calculations. 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();
5  variable[2]=n;//instrument
6  while(i<n){
7       result=calculate(n);
8       i=i+1;
9  }
```

2) Branches: Branches can lead to different execution paths, which may have significantly different execution time. For example, code fragment 2 shows a common process that examines whether a file is successfully opened. If successful, the program will 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 executed. Traditional profiler can obtain statistic probability of branch jump result from sample executions, but it is difficult to predict the jump result in a particular execution. Thus we insert instrumentation code after the condition statement of branches, and record their results as runtime features.

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

3) Loops: Loops are usually the main calculation kernel of a program. The amount of iterations of loops has a direct impact and relationship with the performance. Code fragment 3 demonstrates an example of inserting instrumentations to

count the number of of iterations of loops, including nested loops.

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

4) MPI communications: We do not need to describe the communication behaviors of a parallel program precisely, as long as we can characterize the relationship between communication and performance. Thus 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.

Other approaches can also measure the communication characteristics of a program, like benchmarking or communication tracing. However, benchmarks cannot capture the detailed information, like data movement size, communication type, communication group of a certain MPI communication function call in an execution. Communication tracing can obtain this type of information, but completely tracing MPI events of a parallel program generates traces that need a complex analysis [31].

```
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);
```

C. Model Building

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 consists 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:

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

There exist numerous approaches to solve this regression problem, like ridge regression [9], LASSO [28], artificial neural network (ANN), Gaussian process regression [23], random forest [10], etc. We adopt a random forest approach with an optimization called extremely randomized trees [14]. Random forest is widely applied in classification or regression tasks. Besides the ability 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 [17]. Such a characteristic makes it suitable to model the runtime features we trace from MPI program execution.

Random forest is an ensemble learning method based on CART regression trees [11]. A random forest consists of multiple regression trees. A node in a regression tree selects a feature value θ_i to divide the input data space into two disjoint regions. Finally the regression tree recursively divides the input data space into 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 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. $I()$ is an indicator, which outputs 1 if x belongs to region R_i and 0 otherwise.

To achieve the optimization goal in equation 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 corresponding y of x in region R_i can minimize the MSE [11]. The best division feature can be selected by enumerating all features in the subset and selecting the one with minimum MSE. However, building only one regression tree to divide regions and predict response value often leads to over-fitting. Thus a random forest needs numerous regression trees. Each regression tree fits a random subset in the original features. 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 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 $\Phi()$ is a basis function.

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

For predicting the program execution time, the d-order polynomial expansion function is a reasonable and effective basis function [16], [20], [17], which enumerates the combination of production of original features less than or equal to $\boldsymbol{d}$ orders. It will transform the original $\boldsymbol{m}$ features to $\binom{\boldsymbol{m+d}}{\boldsymbol{d}}$ new features. For example, let $\Phi()$ be a 2-order polynomial expansion function. The transformation of a vector x with two features (a_1, a_2) is:

$$\Phi(x) = (1, a_1, a_2, a_1a_2, a_1^2, a_2^2) \tag{4}$$

This transformation is inspired from the intuition 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, polynomial function can still make an accurate approximation. Experimental results also confirm that this transformation can improve the accuracy of prediction.

D. Features filtration

A straightforward solution of using the collected features as discussed in Section III-B and the modeling technique discussed in Section III-C 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, therefore is strongly desired to filtrate these features to generate a reduced model.

1) Filtration 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 under it. For example, if we set this threshold to be 5%, then in the training phase the features that are generated after 5% of the whole program execution time will be removed. Meanwhile the corresponding instrumentation codes are also removed. After removing these instrumentations, the overhead is significantly decreased too.

2) Filtration by importance: After filtrating features with time threshold, we still need to define and find fewer important features from reserved features. Besides regression, random forest can also analyze the importance of features. As we have described in Section III-C, 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 region) of children nodes is less than that of this node. The extent of MSE reducing can represent the importance of the division feature [10]. Algorithm 1 presents the pseudocode of calculating feature importance using random forest. After calculating importance, we can sort features in the descending order, set a threshold, like 95%, and reserve top features of which accumulative importance is not greater than the threshold.

Algorithm 1 Calculate Importance($forest$)

```
forest.importances = zeros(1..m)
for each tree in forest do
    tree.importances = zeros(m)
    for each node in tree do
        left = node.leftchild
        right = node.rightchild
        tmp = (node.data_size * node.MSE−
        left.data_size * left.MSE−
        right.data_size * right.MSE)
        tree.importances[node.feature] += tmp
    end for
    tree.importances/ = tree.root.data_size
    forest.importances+ = tree.importances
end for
forest.importances/ = forest.n_trees
forest.importances/ = sum(forest.importances)
```

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 approach. New features generated from PCA or SVD are linear combination 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. In addition, since new features are linear combination of original features, they do not have real-world meaning and it is difficult to understand them.

E. Predicting Phase

The predicting phase of our method is simpler than the training phase. After filtrating features and removing corresponding instrumentations, the model is reduced and the instrumented program is transformed into a program fragment. When handling a new input data, the program fragment takes this input and executes partially to fetch feature values. Then the reduced model takes feature values. Each regression tree in

TABLE I
CONFIGURATION OF EXPERIMENTAL PLATFORMS.

Configuration	Platform A	Platform B	Platform C
CPU type	Intel E5-2680v4	Intel E3-1240v5	Intel E7-8860v4
frequency	2.4GHz	3.5GHz	2.2GHz
# cores/node	28	4	144 [1, 32]
mem/node	128GB	32GB	1TB
# nodes	5	40	1
network	100Gbps OPA	100Gbps InfiniBand	100Gbps OPA

the model calculates its response according to feature values. Finally the model outputs the average response of all trees as the predictive execution time of a given input.

IV. EVALUATION

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

A. Experimental Setup

The experiments were conducted on three different platforms denoted as A, B, and C. Table I lists the configuration of each platform. Three applications were tested to predict their execution time under different input parameters. These applications include Graph500 (version 2.1.4) [3], a widely used benchmark focusing on data intensive computing, GalaxSee [2], a parallel N-body simulation program used for simulating the movements of celestial objects, and SMG2000 [4], a parallel semicoarsening multigrid solver for linear systems. Applications are compiled using Intel C/C++ compiler 15.0.0 and run on CentOS 7.3 system. The regression modeling program is written with Python 3.6.1 and scikit-learn library [22]. The MPI library is Intel MPI version 5.0.

B. Feature Filtration

We first evaluate the feature filtration method introduced in Section III-D. In this series of experiments, each application was tested on platform A 100 times with different input parameters to trace runtime features. We filtrate features by time with threshold 5%, and then filtrate by importance with threshold 95% as defined in Section III-D. Table II, III and IV report the experimental results.

Taking GalasSee as an example, and as Table III shows, the complete number of instrumentations of GalaxSee is 357. These instrumentations introduce a total of extra 1404.6% overhead, which means that running the instrumented GalaxSee costs more than ten times of time than the original program. In each run, the trace data generated from instrumentation needs 5,164KB storage on average. After two processes of filtration, only 18 important features are reserved. Storage demand is reduced to 51KB. Their instrumentations only introduce 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 program for HPC users to run their jobs. It will continuously generate runtime feature data and helps the regression be more accurate.

TABLE II
THE IMPACT OF FEATURE FILTRATION IN INSTRUMENTED GRAPH500.

Feature state	# Features	Overhead	Storage (per run)
full	87	65.8%	1,458KB
by time	15	1.3%	85KB
by importance	12	1.2 %	68KB

TABLE III
THE IMPACT OF FEATURE FILTRATION IN INSTRUMENTED GALAXSEE.

Feature State	# Features	Overhead	Storage (per run)
full	357	1404.6%	5,164KB
by time	84	1.1%	615KB
by importance	18	0.5 %	51KB

Figure 2 shows the importance of features in GalsxSee. The meanings of the first five features are the algorithm selection of force calculation, the algorithm selection of numerical integration, the number of cores, a do-while loop counter in the source code nbody.c line 167, and the number of celestial bodies. These information can help developers better tune the application. For example, they can consider the predictive performance of different force calculation algorithms for different input parameters and adopt the better one.

C. Prediction Accuracy

In this section we discuss the prediction accuracy of different machine learning methods for our performance model. The error in our experiments is calculated as $\frac{|y_{predict}-y_{real}|}{y_{real}}$.

We ran each application on each platform 1,000 times with different input parameters. In other words, each modeling task has 1,000 data samples. The input parameters we used for experiments are uniformly and randomly generated from the parameter value range of each application. Part of data samples are randomly selected as the training set while others are the testing set.

Figure 3 presents mean errors on testing set under different ratios of the training data, which illustrates that how many observations the model needs to achieve a low error. The methods we tested include least absolute shrinkage and selection operator (LASSO), ridge regression (Ridge), support vector machine regression (SVR) with radial basis function (rbf) kernel, and random forest (RF). Each method is applied to both the raw form of data and its 3-order polynomial expansion.

TABLE IV
THE IMPACT OF FEATURE FILTRATION IN INSTRUMENTED SMG2000.

Feature State	# Features	Overhead	Storage (per run)
full	648	27.8%	7,821KB
by time	87	0.4%	697KB
by importance	17	0.1 %	151KB

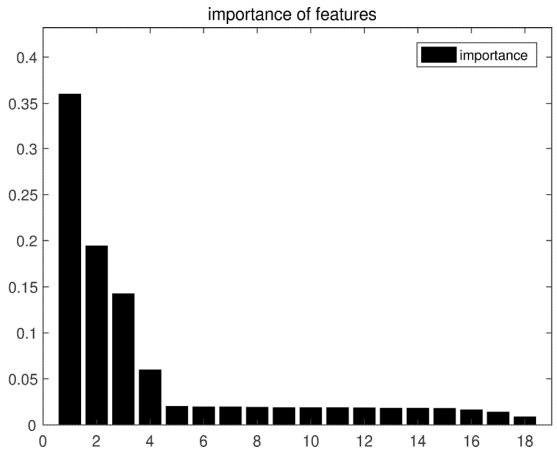

Fig. 2. The importance of features in GalaxSee. Y-axis is the normalized importance, calculated with the random forest regression model. X-axis represents the features of GalaxSee in descending importance order.

Since the mapping relation between the features and the execution time is complicated, two linear regression methods, LASSO and ridge regression, have higher prediction error than nonlinear methods. Using polynomial basis function, these two methods are actually converted to nonlinear methods, and their 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 and RF can achieve lower prediction error. Polynomial expansion provides little benefit to SVR and RF in most cases as shown in Figure 3. Generally, RF has better prediction accuracy. A main possible factor is the impact from 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 numerical meaning. Thus if a regression model considers it as a numerical feature, the result will 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, runtime features of GalaxSee contain categorical features, therefore the superiority of RF is more apparent than that in experiments with other applications.

V. Conclusion

In this paper, we introduce a method to model and predict the performance of parallel programs (MPI programs). We develop a tool to automatically analyze the syntax tree of an MPI program and instrument it, so that we can detect its runtime features related to calculation and communication, without requiring any domain knowledge. We design a strategy to automatically analyze and fit the runtime feature data of an MPI program using random forest technique, thereby we can predict the performance and reveal the factors that significantly affect the performance. Since we adopt a lightweight instrumentation and further reduce them by two filtration processes, the overhead of instrumentation is low and less storage for trace data is demanded.

A limitation of our current method is that features do not capture the affinity information in MPI communications, like the mapping of MPI processes to nodes and cores. Since inner-node communication is faster than intra-node communication, the ignorance of affinity can have a negative effect on the prediction accuracy. However, there are numerous possible mappings. Taking the affinity information into consideration needs a much more complex model and more training data. In future, we will further investigate how to capture the communication features more comprehensively and further improve the accuracy of our model.

Acknowledgments

Guangzhong Sun is the corresponding author of this paper. This study is supported by NSF of China (grant number: 61772485). 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.graph500.org/referencecode.
[4] The smg2000 benchmark code. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[9] C. M. Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.
[10] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
[11] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.
[12] 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.
[13] 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.

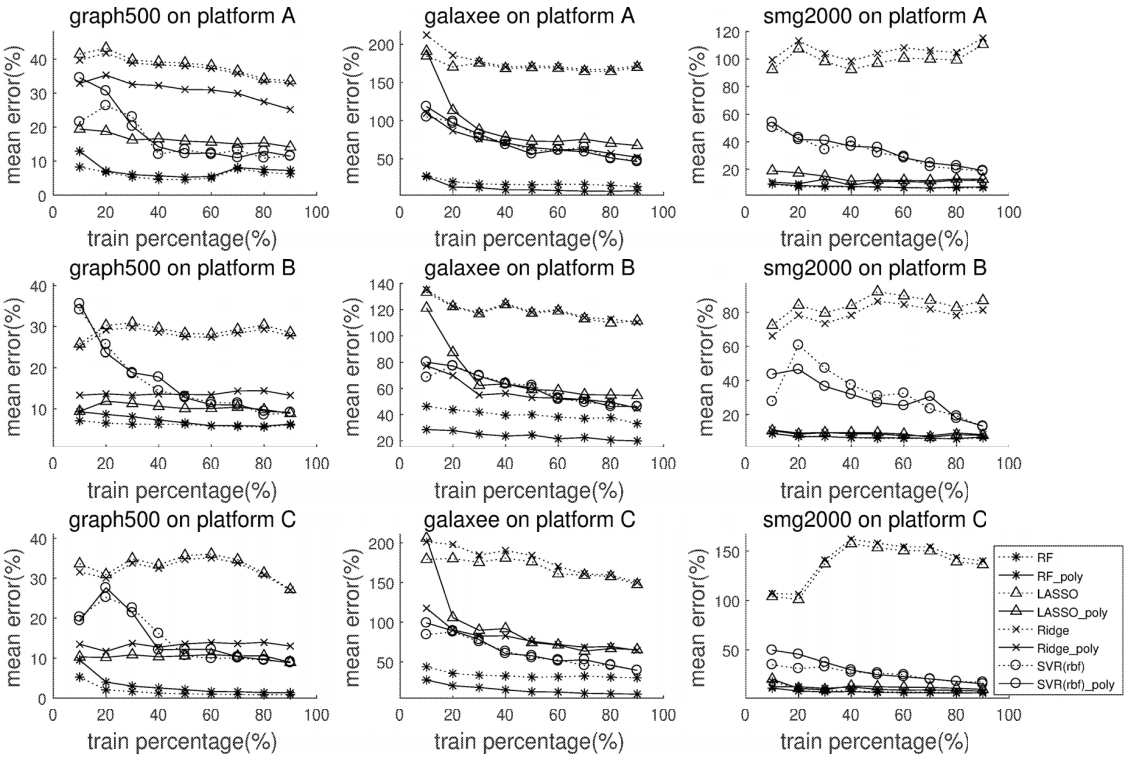

Fig. 3. The prediction error of different machine learning methods with varying applications and platforms.

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

[15] 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.

[16] 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.

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

[18] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*. Springer, 2005.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] C. E. Rasmussen. Gaussian processes for machine learning. 2006.

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

[25] K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches. *Concurrency and Computation: Practice and Experience*, 19(17):2219–2235, 2007.

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

[27] 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. ACM, 1999.

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

[29] 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.

[30] 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.

[31] 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.