Multi-Parameter Performance Modeling Based on Machine Learning with Basic Block Features

Meng Hao*, Weizhe Zhang*†, Yiming Wang*, Dong Li‡,Wen Xia*†, Hao Wang§, Chen Lou*
*School of Computer Science and Technology, Harbin Institute of Technology, China
†Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China
‡Department of Electrical Engineering and Computer Science, University of California, Merced
§Department of Computer Science, Norwegian University of Science and Technology, Norway
{haomeng, wzzhang, yimingw, xiawen}@hit.edu.cn, dli35@ucmerced.edu, hawa@ntnu.no, loucchen@gmail.com

***Abstract*—Considering the increasing complexity and scale of HPC architecture and software, the performance modeling of parallel applications on large-scale HPC platforms has become increasingly important. It plays an important role in many areas, such as performance analysis, job management, and resource estimation. In this work, we propose a performance modeling and prediction framework called SmartPred, which utilizes basic block frequencies as features and uses machine learning algorithms to automatically construct multi-parameter performance models with high generalization ability. To reduce the prediction overhead, we propose some feature-filtering strategies to reduce the number of features in the training stage and build a serial program called BBF collector for each target application to quickly collect feature values in the prediction stage. We demonstrate the use of SmartPred on the TianHe-2 supercomputer with six parallel applications. Results show that SmartPred with SVR achieves better prediction than other input parameter-based modeling methods. The average prediction error and average standard deviation of prediction errors of SmartPred are 8.42% and 6.09%, respectively. In the prediction stage, the average prediction overhead of SmartPred is less than 0.13% of the total execution time.**

***Index Terms*—Performance Prediction, Parallel Application, Basic Block Feature, Machine Learning**

I. Introduction

The demand for computing resources in science and engineering is increasing, so supercomputer performance is continually being upgraded from petascale to exascale. Accordingly, the complexity of HPC architecture and software is rising dramatically [1]. As a result, the efficiency and scalability issues for HPC systems and applications become increasingly prominent. Performance modeling, which can be used to understand and predict the performance of parallel applications on HPC systems, is a main concern in the HPC community to assist in solving these prominent issues.

Building multi-parameter models to predict the execution time of parallel applications with any inputs and at any scale yields many benefits, and plays an important role in many areas, such as performance analysis and tuning [2], [3], job management and scheduling [4], [5], and resource estimation [6], [7]. However, many factors affect the application performance (i.e. execution time), including system architectures, system middlewares, applications, and input parameters. For example, the contention for shared resources, such as the intra-node shared cache and memory or the inter-node interconnected network on the large-scale cluster, can result in complex program behaviors. Due to the complex interaction of these factors, building a performance model, especially the multi-parameter model that can be used to accurately predict the performance of parallel applications with any inputs and at any scale, is a very challenging task.

Considerable methods have been utilized in previous research to address this issue, such as trace-driven simulators [8], [7] or analytical modeling methods [9], [10]. These methods often consume enormous resources, including both man hours and machine allocation [11], or need domain experts for the analysis of the system and algorithm in depth, which limits their accessibility to users, who are not familiar with target systems or applications in details. For a good usability of these performance tools, machine learning is widely used in performance modeling [6], [12]. These methods treat applications and systems as black boxes and create empirical performance models automatically through machine learning algorithms. However, the accuracy of machine learning performance models depends on the representativeness of the training data set. Existing machine-learning-based modeling methods consider input parameters as features and usually have limited generalization ability. Hence, they cannot accurately predict the performance of parallel applications when inputs are out of range of the training set.

The use of machine learning for building performance models can hide the details of the underlying architecture and application and capture complex relationship between multiple parameters and performance without human intervention. However, two main challenges are faced with the use of machine learning to build performance models as follows:

(1) *Selection of the appropriate features for the performance modeling of parallel applications*. Data and features can bound the upper limit of the prediction accuracy of machine learning, and machine learning models and algorithms just approach this upper limit. Therefore, appropriate features can be utilized to improve the generalization ability of performance models.

(2) *Quick measurement of values of the selected features, especially in the prediction stage*. This process is related to the usability of the performance modeling method.

To address the above challenges, we present SmartPred, a *novel performance modeling and prediction framework for an accurate prediction of the performance (wall time) of MPI applications with any inputs and at any scale.* SmartPred considers *basic block frequencies* (BBF) as features and uses a machine learning algorithm, such as support vector regression, to automatically construct multi-parameter performance models with high generalization ability.

The basic block (BB) is a unit of sequentially executed instructions having a single entry and a single exit point. Its BBF represents the execution count of a corresponding code segment, which can characterize the complexity of program behaviors better than the input parameters. Hundreds of or thousands of BBs are utilized in actual parallel application, and their BBFs are runtime features, which usually need to be obtained by executing the target parallel application completely on HPC systems. To reduce the prediction overhead, SmartPred adopts some *feature filtering strategies* to reduce the number of features in the training stage and builds a serial program called the *BBF collector* for each target application in the prediction stage. Then, the BBF feature values can be collected quickly by executing the collector on one node of any platform instead of executing the original parallel application on large-scale HPC systems.

The specific contributions of this paper are summarized as follows:

(1) We propose a novel framework to build multi-parameter performance model with higher generalization ability than traditional input parameter-based modeling methods [6], [12], and predict the performance of parallel application with different inputs at different scales. Our experiments with six different parallel applications on the TianHe-2 supercomputer showed an average prediction error of 8.42%, and average standard deviation of prediction errors of 6.09%.

(2) We are the first to use the basic block frequencies as training features and propose corresponding feature filtering strategies to reduce the number of features. These features are necessary to support automated modeling and prediction with machine learning algorithms.

(3) We developed a tool to automatically construct the BBF collector for the target parallel application. The collector is a serial program, which can be used to collect BBF feature values without executing the original parallel application. The average overhead of the collector was only less than 0.13% of the original execution.

The rest of this paper is organized as follows: Section II introduces some background knowledge and motivates the paper with an example. Section III presents the overview of SmartPred. Section IV and Section V describe the training and prediction stages of SmartPred, respectively. The experimental results are presented in Section VI. Finally, Section VIII provides some conclusions.

II. Background and Motivation

Previous machine learning methods [13], [6], [3] often consider application input parameters as features, run target parallel application multiple times with varying input parameters to collect training data, and then use machine learning algorithms, such as Random Forest (RF), Ridge Regression (Ridge) and SVR, to predict the execution time of parallel applications. However, only considering the input parameters as features is not enough. The generated performance model often has limited generalization ability, and can not accurately predict the performance of parallel application with unseen inputs. In this section, we considered a compact parallel application (Sweep3D) to illustrate above points.

We collected a sample data set by choosing a collection of points spread across the input parameter space and then obtain the performance results for each sample on the actual HPC system. We then grouped the original data set proportionately into the following parts:

(1) Training set: Samples used to build the performance model through machine learning algorithms. We applied cross-validation to compare the performance of current models and chose the model with the best performance.

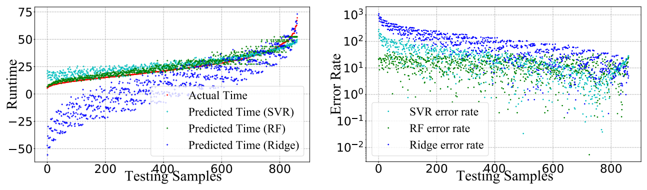
(2) Testing set: Samples used for reporting the final effects of models and were not utilized in model training. We used this set to verify the generalization ability of performance models.

Table I shows the selected input parameters of Sweep3D and their corresponding value ranges. The total problem grid size of this application was determined based on three parameters: IT_G, JT_G and KT. The number of processes (represented by #P) used is equal to the product of NPE_I and NPE_G. In this table, the testing sets contain samples of which the feature values are out of range of the training sets, especially the number of processes #P. We used this testing set to validate the generalization ability of existing performance models.

We first used these input parameters as features, used three different machine learning algorithms (SVR, RF and Ridge) to train performance models, and evaluated the performance of each model with same testing set. To compare the performance of different models, we reported the mean absolute percentage error (MAPE) for each machine learning method [14], which indicates how close the prediction is to the actual value.

TABLE I: The input parameters of Sweep3D.

Input Parameter	Training Set	Testing Set
IT_G	300, 400, $\cdots$, 700	300, 400, $\cdots$, 800
JT_G	300, 400, $\cdots$, 700	300, 400, $\cdots$, 800
KT	300, 400, $\cdots$, 700	300, 400, $\cdots$, 800
#P	16, 48, $\cdots$, 176	208, 224, $\cdots$, 256



(a) Predicted runtimes (b) Prediction errors

Fig. 1: The prediction results of Sweep3D by using different machine learning algorithms with the input parameter features.

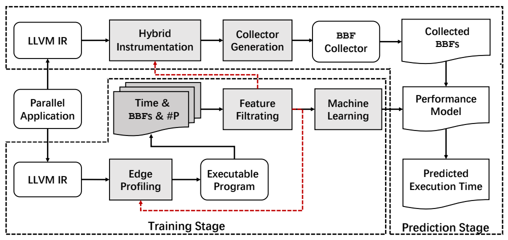


Fig. 2: The framework overview of SmartPred.

Figure 1 presents the prediction results of different machine learning algorithms with the selected features. The mean absolute percentage errors for these different machine learning algorithms (SVR, RF, and Ridge) were 25.80%, 14.17% and 109.83% respectively. Results showed that *when using the input parameters as features, these machine learning algorithms cannot accurately predict the performance of parallel applications*, because after receiving the input parameters, the application underwent a series of calculations. Subsequently, the input parameters and application performance had complicated non-linear relations. The entire model space cannot be traversed to obtain an optimal mapping between input parameters and application performance.

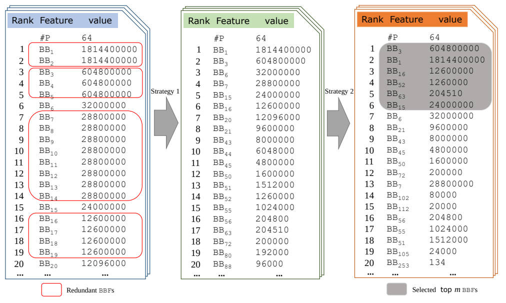
Therefore, to enhance the machine learning-based performance modeling of parallel applications, we replaced the input parameters with some runtime features (BBFs) and build more generalized performance models. In comparison with the input parameters, BBFs representing the execution count of corresponding code segments can better reflect the application performance, and both have simple relations. When using BBFs as training features, these machine learning algorithms can be utilized to establish performance models with high generalization ability.

III. Overview

In this section, we provide an overview of our automatic performance modeling and prediction framework called SmartPred based on runtime features, namely, BBF features.

As shown in Figure 2, SmartPred includes the training and prediction stages. The training stage is used to collect data and build the performance models. The prediction stage is used to effectively collect the BBF values for the parallel application with new inputs and at new scale, resulting in a predictive execution time by using the trained model.

In the training stage, SmartPred considers BBFs as features and applies *edge profiling*, which is a lightweight instrumentation method with low storage demand to instrument the LLVM's intermediate representation (IR, for short) of the parallel application. Then, the instrumented version is tested on the target HPC system for multiple times with varying values of input parameters to collect data, including values of BBFs, number of processes (#P), and execution time (time) for training models. Besides, SmartPred adopts some *feature filtrating strategies* to decrease the number of BBF features, thereby reducing training time of machine learning algorithms and the extra overhead of instrumentation.



Rank	Feature	value
	#P	64
1	BB_1	1814400000
2	BB_2	1814400000
3	BB_3	604800000
4	BB_4	604800000
5	BB_5	604800000
6	BB_6	32000000
7	BB_7	28800000
8	BB_8	28800000
9	BB_9	28800000
10	BB_{10}	28800000
11	BB_{11}	28800000
12	BB_{12}	28800000
13	BB_{13}	28800000
14	BB_{14}	28800000
15	BB_{15}	24000000
16	BB_{16}	12600000
17	BB_{17}	12600000
18	BB_{18}	12600000
19	BB_{19}	12600000
20	BB_{20}	12096000
...	...	...

Rank	Feature	value
	#P	64
1	BB_1	1814400000
2	BB_3	604800000
3	BB_6	32000000
4	BB_7	28800000
5	BB_{15}	24000000
6	BB_{16}	12600000
7	BB_{20}	12096000
8	BB_{21}	9600000
9	BB_{43}	8000000
10	BB_{44}	6048000
11	BB_{45}	4800000
12	BB_{50}	1600000
13	BB_{51}	1512000
14	BB_{52}	1260000
15	BB_{55}	1024000
16	BB_{56}	204800
17	BB_{63}	204510
18	BB_{72}	200000
19	BB_{80}	192000
20	BB_{88}	96000
...	...	...

Rank	Feature	value
	#P	64
1	BB_3	604800000
2	BB_1	1814400000
3	BB_{16}	12600000
4	BB_{52}	1260000
5	BB_{63}	204510
6	BB_{15}	24000000
7	BB_6	32000000
8	BB_{21}	9600000
9	BB_{43}	8000000
10	BB_{45}	4800000
11	BB_{50}	1600000
12	BB_{72}	200000
13	BB_7	28800000
14	BB_{102}	80000
15	BB_{112}	20000
16	BB_{56}	204800
17	BB_{55}	1024000
18	BB_{51}	1512000
19	BB_{105}	24000
20	BB_{253}	134
...	...	...

Fig. 3: The schematic of feature filtering process.

In the prediction stage, SmartPred uses *hybrid instrumentation* method, which combines static profiling with dynamic profiling to instrument the IR of parallel application. Then, it constructs the BBF collector which, is a serial program for the collection of the BBF values of target parallel application with new inputs and at new scale without executing it on the target largescale HPC system. Hence, in this stage, we simply need to run a serial program on one node of any platform (even an ordinary PC or laptop) to achieve the performance prediction.

IV. Training Stage

A. Edge Profiling

This part aims to instrument original parallel applications and then collect BBFs by executing the instrumented version. However, we do not need to insert a counter into each BB to measure its precise execution frequency. Edge profiling, one of the optimized dynamic profiling techniques, is widely used in feedback-directed optimization to improve the program runtime performance. In the training stage, we used this technique to instrument parallel applications.

Edge profiling involves the insertion of a counter for each selected edge in the program. Then, the counter is incremented each time the corresponding edge executes. Counters are usually placed in either the source or target BB. However, for the critical edge, which is neither the only edge leaving its source BB, nor the only edge entering its target BB, it needs to be split by inserting a new BB between its source and target; then, the counter can be placed in this BB. Ball et.al.[15] used the maximum spanning tree to optimize the placements of these counters.

Executing the instrumented program will generate a profile, which provides edge frequencies for subsequent calculation of all BBFs.

B. Feature Filtration

Many BBs are present in the IR of an actual parallel application. For example, the IR of Sweep3D contains 763 BBs, and we cannot use all of them as features, because the use of too many features will generate a fairly complex model and introduce significant training overhead. Therefore, we propose two feature filtering strategies to reduce the number of BBF features in this section.

The first strategy involves the remove redundant BBF features. As shown in Figure 3, after getting one profile, which includes BBs and the corresponding frequencies, we first sorted these BBs in the descending order of frequency. Considering the control flow among BBs, regardless of how input parameters change, some BBFs remain the same. In these cases, in each BB group with same frequency, we retained one and removed the other redundant BBFs. Some redundant features had variance if zero, in which the values of these features are same in all samples. Subsequently, we further detected and removed these zero-variance BBF features.

Not all the rest of BBFs are appropriate as the training features, because an excessive number of irrelevant features (noise information) may degrade the generalization ability of models and lead to overfitting. Besides, due to the large number of BBs, exhaustive feature selection process is not ideal. To effectively choose the BBF features with strong correction with the final performance of application, we used the univariate feature selection method, which quantifies the correlation between each BBF feature and the corresponding execution time by using Pearson correlation coefficient (r) [13]. Pearson correlation coefficient is a measure of the linear correlation between two variables, which provides a good understanding of data. Therefore, instead of considering all the rest of BBFs as training features, we proposed the second strategy, which involves sorting the rest of BBFs in descending r order and choosing BBFs ranked in the top m places as training features. The value of m can be determined by users according to the actual situation.

Once the critical BBF features have been selected using these feature filtering strategies, we can optimize the instrumentation method in the prediction stage by ignoring unselected BBs and effectively reduce instrumentation overhead.

C. Model Generation

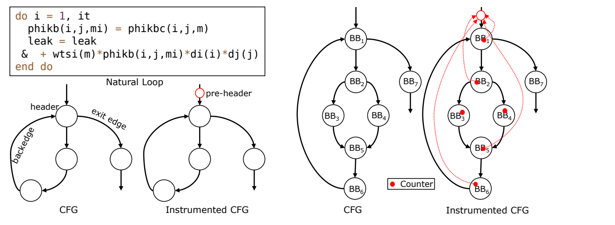
After collecting the values of BBF runtime features via edge profiling and using two filtrating strategies to reduce the number of these features, we extrapolated mappings between runtime features and application execution time to allow the use of such mapping for any new input and scale to make a prediction of timing. It can be treated as a multivariate nonlinear regression problem.

Formally, we used symbol D as the training set, which includes n samples as follows:

$$D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_i, y_i), \cdots, (x_n, y_n)\}$$

Each pair (x_i, y_i) represents a sample in the training set. In each sample, $x \in \mathbb{R}^d$ is a d-dimensional feature vector consisting of $m + 1$ features, which include m BBFs and the number of processes #P. The input parameter #P is included in the feature vector, because it is closely related to the communication performance of parallel application. Due to the large difference (several orders of magnitude) among the frequency values of some BBs, we considered logarithm of each BBF, that is

$$x = \{log(p), log(b_1), log(b_2), \cdots, log(b_j), \cdots, log(b_m)\}$$



```
do i = 1, it
  phikb(i,j,mi) = phikbc(i,j,m)
  leak = leak
 &  + wtsi(m)*phikb(i,j,mi)*di(i)*dj(j)
end do
```

Fig. 4: Some CFG examples about our hybrid Instrumentation method.

where p is the value of #P, b_j represents the value of each selected BBF feature. Similarly, $y \in \mathbb{R}$ indicates the logarithm of corresponding execution time.

Our regression problem aims to determine a function $f^* : x \to y$, which can minimize the mean square error between the predictive value and actual execution time in the n samples:

$$f^* = \arg\min_{f \in \mathcal{F}} (\frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2) \quad (1)$$

where $\mathcal{F}$ represents the hypothesis space.

Numerous algorithms can be used to solve this regression problem, such as RF, SVR, and Ridge. In this paper, we used all these algorithms to obtain the performance models and then validate the effectiveness of our BBF feature-based performance modeling method.

V. Prediction Stage

A. Hybrid Instrumentation

In comparison with the actual execution of parallel applications (e.g., Sweep3D), *edge profiling* introduces approximately 20% extra instrumentation overhead. Therefore, in the prediction stage, to collect the feature vector of application with new input and at new scale more effectively, we proposed a more lightweight hybrid instrumentation method, which combines static profiling with dynamic profiling to instrument the IR of the parallel application.

Unlike *edge profiling*, which inserts a counter for almost each edge, our hybrid instrumentation method only inserts counters for selected BBs and then records the values of these BBF features, which can significantly reduce the number of counters required. Besides, many loops are usually found in scientific parallel applications, whose loop trip count (LTC, for short) can be determined before the execution of the loop. In this paper, this type of loop is called a *natural loop*, which usually consists of the following:

(1) a header basic block, which is the basic block where the decision is made whether or not to execute the loop;

(2) a backedge, which goes from the loop back into the header; and

(3) an exit edge, which is traversed in case the loop is not executed any more.

Figure 4a shows a code snippet of Sweep3D, which includes one natural loop and the corresponding control flow graph (CFG). We can insert a new BB called preheader before

Algorithm 1 Hybrid instrumentation algorithm

```
Input: The IR of target program $\mathcal{P}$
Output: The instrumented version of IR
 1: Get the selected BBF feature set $\mathcal{S}$ in the training stage;
 2: Create counter array $\mathcal{C}$ in $\mathcal{P}$, and initialize $\mathcal{C}$ with zeros;
 3: $index := 0$;
 4: for all basic blocks $b$ in $\mathcal{S}$ do
 5:     Get the loop $l$ which $b$ belongs to;
 6:     Get the head block $h$ of backedge in loop $l$;
 7:     if $l$ is a natural loop and $b \gg h$ then
 8:         Construct a preheader block $p$ before the header;
 9:         Get the LTC related values: start, end, stride;
10:         Add code before the terminator instruction of $p$ such that $\{\tau = \frac{\texttt{end}-\texttt{start}}{\texttt{stride}} + 1\}$ to calculate LTC of $l$;
11:         Add code before the terminator instruction of $p$ such that $\{\mathcal{C}[index] += \tau\}$ is executed when $p$ is traversed;
12:     else
13:         Add code to $b$ such that $\{\mathcal{C}[index]++\}$ is executed when $b$ is traversed;
14:     end if
15:     $index++$;
16: end for
17: Add code to the end of $\mathcal{P}$ that writes $\mathcal{C}$ to file;
18: return The instrumented IR;
```

the header of loop and move BB counters located in natural loops into preheader block. This method can further reduce the number of counters accessing and updating.

However, not all BB counters inside natural loops can be moved into the preheader. Figure 4b shows one example about the changing of insertion position of BB counters in a natural loop containing branches. The counters of BBs inside branches cannot be moved into preheader, because the branch target can only be determined during execution.

Definition 1. (DOMINANCE). *In a control flow graph with entry node b_0, node b_i dominates node b_j if and only if every path from the b_0 to b_j must go through b_i, which written as $b_i \gg b_j$. By definition, each node dominates itself, that is, $b_i \gg b_i$.*

We used the notion of *dominance* to determine whether BB counters can be moved into the preheader. In the natural loop, the number of backedge traversed is equal to the loop trip count, thereby allowing counter of backedge's head block to be moved into preheader. In general, according to the definition of *dominance*, for each BB inside the natural loop, if it dominates the backedge's head, its counter can also be moved into preheader. For example in Figure 4b, the BBs whose counters can be moved into preheader include $\texttt{BB}_1$, $\texttt{BB}_2$, $\texttt{BB}_5$ and $\texttt{BB}_6$, because they all dominate the backedge's head block $\texttt{BB}_6$.

The detailed process of hybrid instrumentation method is shown in Algorithm 1.

B. Collector Generation

In the prediction stage, executing the instrumented parallel application to collect the values of BBF features is time-consuming and requires a significant amount of computing resources, especially for parallel applications at large scale. In this section, we generated one BBF collector, which is a serial program for each target parallel application and executed the collector instead of the original application to collect BBF values.

Algorithm 2 Collector Generation Algorithm.

```
Input: The instrumented IR of a parallel program
Output: The IR of generated collector
 1: Contain all initialization and instrumentation related codes;
 2: Construct the program call graph (PCG) based on IR;
 3: for all functions $f$ in Post-Order over PCG do
 4:     Construct the control flow graph (CFG) of $f$;
 5:     for all basic blocks $b$ in Post-Order over CFG do
 6:         do
 7:             for all instructions $i$ in Bottom-Top order over $b$ do
 8:                 if $i \in$output or is one MPI function call then
 9:                     Remove instruction $i$ from $b$;
10:                     Perform the dead code elimination on $b$;
11:                     Break;
12:                 end if
13:             end for
14:         while $b$ is changed
15:     end for
16: end for
17: Remove unused functions and their function declarations;
18: return the IR of generated collector;
```

We divided one instrumented IR of parallel application into the following parts parts:

(1) *Initialization*. This part is used to read the input data, gather execution configuration, and prepare data structures.

(2) *Instrumentation*. This part includes all inserted codes which are responsible for collecting BBF values.

(3) *Computation*. This part is responsible for doing a parallel computation by spawning several processes, which can communicate with one another during computation.

(4) *Output*. This part is used to print the final results of computation to console.

We only aimed to calculate the frequencies of selected BBs without considering the computation results. Hence, we first retained the *initialization*- and *instrumentation*-related codes to guarantee the generated programs normal execution and the accurate recording of BBF values. Then, we directly removed useless *output*-related codes from IR. Besides, to generate one self-contained BBF collector, which involves a serial program and does not depend on any MPI libraries, we also need to remove the MPI function calling codes that belongs to the *computation* part.

After removing *output*-related codes and MPI function calling codes, many dead codes whose results are never used in any other computation would arise. We performed dead code elimination [16] to remove these dead codes from IR, which can shrink the IR form and lead to a smaller executable program and faster execution. The detailed process of collector generation is shown in Algorithm 2.

VI. Evaluation

We have implemented our SmartPred framework, which includes *edge profiling*, *hybrid instrumentation* and *collector generation*, as IR-to-IR transformations in the Clang/LLVM compiler framework [17]. We selected three machine learning algorithms, including RF, Ridge and SVR, with radial basis function kernel to demonstrate the effectiveness of our BBF-based method. These three machine learning algorithms have various degrees of complexity and are widely used for handling modeling problems [18], [19], [20].

TABLE II: The input parameters of selected benchmarks.

App	Description	Input Parameter	Type
Sweep3D	Neutron transport application	`IT_G,JT_G,KT,#P`	Integer
LULESH	Shock hydrodynamics application	side length `(-s),#P`	Integer
NPB SP	Scalar penta-diagonal solver	`problem_size,#P`	Integer
NPB BT	Block tri-diagonal solver	`problem_size,#P`	Integer
NPB LU	Lower-upper Gauss-Seidel solver	`nx,ny,nz,#P`	Integer
NPB EP	Embarrassingly parallel random number generator	`m,#P`	Integer

A. Platforms and Benchmarks

We use the TianHe-2 Supercomputer in National Supercomputer Center in Guangzhou as the experimental platform to evaluate our SmartPred framework. It contains 16,000 compute nodes, each with two 12-cores 2.2 GHz Intel Xeon E5-2692 processors and 64 GB of memory. These nodes are interconnected via the TH Express-2 network. We evaluate SmartPred with four NPB programs (SP, BT, LU and EP)[21] and two full real-world parallel applications (ASCI Sweep3D [10] and LULESH [22]). The overview of all test programs and their critical input parameters are shown in Table II.

Previous works [13], [6], [18] usually generate samples uniformly and randomly from the parameter value range of each application. Then, part of data samples are randomly selected as the training set, whereas the others are selected as testing set. By comparison, our testing sets contain samples of which the feature values are out of range of the training sets. This process was done to improve the accuracy of evaluating the generalization ability of performance models.

B. Performance Prediction Results

Table III shows two sets of features, one (`INPUT`) only contains input parameters, and the other one (`RUNTIME`) contains `BBF` features, which are selected using our feature filtering method. Notably, the `RUNTIME` set contains `#P`, and this feature is closely related to the communication performance of parallel application. We choose these two feature sets and used three different machine learning algorithms (SVR, RF and RR) to train performance models, and evaluate the performance of each model with the same testing set.

We calculate the mean absolute percentage errors for the prediction results of each application in Table III. Results showed that compared with input parameter features, the use of `BBF` as features can significantly improve the prediction accuracy of machine learning algorithms. Moreover, when using only input parameter features, the RF can obtain the best prediction results. By contrast, SVR is better than RF when using `BBF` features. Its average error is 8.42%, and the average standard deviation of error is 6.09%.

C. Comparison with other methods

In this section, we performed some experiments to compare the performance of SmartPred with two other classical input parameter-based modeling methods, namely, Barnes' method [6] and Hoefler's method [12]. Both methods are based on regression and use linear models to construct the relationship between the input variables and the observed execution time by varying the values of the input variables (x_1,x_2,. . . ,x_n) on the instrumented runs.

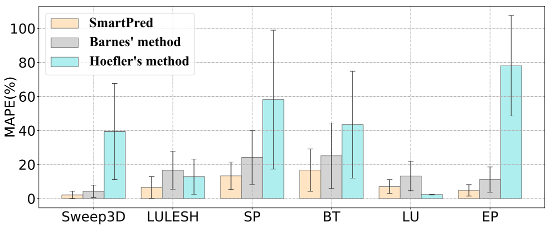


Fig. 5: Prediction results with our SmartPred compared with that with the Barnes' method and the Hoefler's method on the *TianHe-2* supercomputer. Error bars indicate standard deviation which is used to quantify the variance of prediction errors.

The Barnes' method assumes the following model form:

$$\begin{aligned} log_2(T) =& \beta_1 log_2(x_1) + \beta_2 log_2(x_2) + ... \\ &+ \beta_n log_2(x_n) + g(q) + error \end{aligned}$$

where $g(q)$ can be either a linear function or quadratic function for the number of processors q, such as

$$g(q) = \gamma_0 + \gamma_1 log_2(q), g(q) = \gamma_0 + \gamma_1 log_2(q) + \gamma_2(log_2(q))^2$$

In this experiment, we predicted the performance with these two forms of $g(q)$ function and reported the best results.

By comparison, Hoefler's method implements the PEMOGEN framework, which uses complex model form called EPMNF, as follows:

$$f(P) = \sum_{i=1}^{|P|} \sum_{k=1}^{m} c_{ik} \cdot p_i^{j_{ik}} \cdot log_2^{l_{ik}}(p_i)$$

where P is the parameter set. A possible assignment of j_{ik} and l_{ik} is called a *model hypothesis*. In this method, users need to provide the candidate sets of j_{ik} and l_{ik} to trade off between the model generation overhead and prediction accuracy. The coefficients c_{ik} of all hypotheses are determined via regression. This method chooses the hypothesis with the smallest prediction error to determine the most likely performance model. In this experiment, we expanded the hypothesis sets used in [12] and used the values of $j_{ik} = \{-1, 0, \frac{1}{3}, \frac{1}{2}, 1, \frac{3}{2}, 2, \frac{5}{2}, 3\}$ and $l_{ik} = \{0, 1, 2, 3\}$. Additional values into the hypothesis sets only increases the model generation overhead but does not help in improving the prediction accuracy.

Figure 5 demonstrates the prediction results with SmartPred, Barnes' method, and Hoefler's method for the six parallel applications on the *TianHe-2* supercomputer. The average prediction errors of these three modeling methods are 8.42%, 15.74% and 39.03%,respectively. No matter the prediction errors or the standard deviations, SmartPred can obtain lower values than the two other methods. Hence, compared with the traditional input parameter-based modeling methods, our machine learning-based method with `BBF` features can build

TABLE III: The mean absolute percentage errors and corresponding standard deviations for three different machine learning methods with different feature sets.

App	Feature Set	Features	MAPE			Standard Deviation		
			SVR	RF	Ridge	SVR	RF	Ridge
Sweep3D	INPUT	IT_G, JT_G, KT, #P	25.80%	14.17%	109.83%	31.15%	9.24%	136.82%
	RUNTIME	BB$_3$, BB$_1$, BB$_{16}$, BB$_{52}$, BB$_{63}$, BB$_{15}$, #P	2.13%	4.18%	4.71%	2.20%	3.47%	4.33%
LULESH	INPUT	-s, #P	163.33%	6.58%	107.43%	190.97%	3.09%	88.44%
	RUNTIME	BB$_{21}$, BB$_{22}$, BB$_{26}$, BB$_{27}$, BB$_{34}$, BB$_{36}$, #P	6.52%	6.16%	13.01%	6.44%	3.99%	9.89%
NPB SP	INPUT	problem_size, #P	311.77%	26.67%	1264.74%	701.13%	23.95%	2922.95%
	RUNTIME	BB$_{25}$, BB$_{28}$, BB$_{39}$, BB$_{46}$, BB$_{51}$, BB$_{56}$, #P	13.33%	19.12%	26.97%	8.12%	16.61%	18.01%
NPB BT	INPUT	problem_size, #P	272.88%	29.13%	1722.82%	627.30%	47.39%	3899.25%
	RUNTIME	BB$_{13}$, BB$_{30}$, BB$_{31}$, BB$_{48}$, BB$_{49}$, BB$_{50}$, #P	16.71%	22.29%	26.84%	12.45%	29.22%	27.11%
NPB LU	INPUT	nx, ny, nz, #P	24.11%	14.22%	12.90%	2.85%	4.80%	10.11%
	RUNTIME	BB$_4$, BB$_{11}$, BB$_{14}$, BB$_{28}$, BB$_{51}$, BB$_{53}$, #P	7.03%	30.14%	12.18%	4.04%	4.43%	9.31%
NPB EP	INPUT	m, #P	458.86%	28.28%	553.33%	763.56%	19.51%	920.18%
	RUNTIME	BB$_2$, BB$_3$, BB$_4$, BB$_9$, BB$_{10}$, BB$_{11}$, #P	4.80%	10.00%	3.64%	3.31%	8.04%	2.86%

TABLE IV: The average prediction overhead of SmartPred as well as that of the original execution.

App	Original Overhead (Core Hour)	Prediction Overhead (Core Hour)	Proportion
Sweep3D	1.5574	0.0041	0.2633%
LULESH	35.4272	0.0441	0.1245%
NPB SP	28.5981	0.0177	0.0619%
NPB BT	22.9459	0.0150	0.0654%
NPB LU	8.0889	0.0028	0.0346%
NPB EP	7.1099	0.0129	0.1814%
Average	–	–	0.1219%

generalized performance models and significantly improve prediction accuracy.

Among the three methods, the experimental results of Hoeflers method are the worst. In addition to the limitation of input parameter features, this condition occurred because the Hoeflers method classifies the whole program into several loop kernels and models these kernels by using regression. This method is helpful for modelling some kernels, which have simple performance behaviors. However, when modeling the execution time of whole parallel applications with more complex performance behaviors, this method has limited effectiveness [23].

D. Performance Prediction Overhead

Notably, when predicting the performance of a parallel application, we only need to execute the corresponding serial collector to collect `BBF` values instead of executing the original parallel application. The generated data contains only `BBF` values of several basic blocks, and its storage overhead is negligible. Therefore, in this section, we primarily evaluated the execution overhead of the collector in the prediction stage.

Computational resources on supercomputers are billed in core hours. Hence, in this experiment, the prediction overhead of SmartPred was also expressed in core hours. Table IV shows the comparisons of consumed core hours of SmartPred when predicting performance of six selected applications with that of the original parallel application execution. All overheads of SmartPred performed on six applications are substantially below the overheads of the original application execution. Moreover, the average overhead only accounts for 0.1219% of the original execution. Hence, SmartPred can help HPC users to predict the performance of parallel applications efficiently, because the `BBF` collector is an independent serial program, which can be executed with only one node or core. We also optimized the collector by reducing the number of inserted counters and eliminating many dead codes, which further improved its performance.

Note that LLVM IR [17] is quite portable over the various architectures. Therefore, in the prediction stage, we can compile and execute the `BBF` collector on one node of any platform (even an ordinary PC or laptop) to collect feature values instead of executing it on one node of target platform. Hence, the use of SmartPred can easily achieve cross-platform performance prediction.

VII. Related Work

Many works related to the performance prediction of parallel applications have been proposed, which can be divided into two categories: model- and trace-based methods.

Model-based methods: These methods commonly require the construction of performance models for the computing systems and programs. The execution time is predicted by calculating and analyzing these models [6], [13], [18], [12]. These methods treat applications and systems as black boxes and create empirical performance models automatically through machine learning algorithms. However, these empirical modeling methods only apply to certain types of applications (e.g., strong-scaling or weak-scaling) or need to execute original parallel applications for the collection of some information in the prediction stage, which consumes enormous time and computing resources.

By comparison, SmartPred chooses frequencies of some basic blocks as new features and automatically construct multi-parameter performance models with high generalization ability. In the prediction stage, instead of executing original parallel applications, SmartPred builds a serial collector to collect `BBF` values, which can significantly reduce the prediction overhead.

Trace-based methods: Trace-driven methods, which are frequently used in simulators [8], [7] and benchmark generation tools [24], [25], can capture detailed performance behavior and model the performance of parallel programs automatically. However, trace-driven methods have some limitations. First, due to the complexity of hardware and software, building one simulator often consumes enormous huge man hours and machine allocation [11]. Second, traces contain large amounts of performance-related data, thereby requiring large amounts of storage. Third, generating traces of large-scale parallel application is very expensive, because it needs to execute original applications on an existing system with corresponding scale processors. Fourth, heavyweight instrumentation used in these methods may affect the behavior of parallel programs.

By comparison, SmartPred uses lightweight instrumentations (edge profiling and hybrid Instrumentation) in collecting profiling data, thereby requiring less storage.

VIII. Conclusion

We proposed the SmartPred, a novel performance modeling and prediction framework based on basic block features, to build multi-parameter performance model with high generalization ability and predict the performance of parallel application with different inputs at different scales on distributed-memory architectures. Results showed that SmartPred can improve the prediction accuracy while greatly reducing the prediction overhead.

The SmartPred is beneficial for both HPC users and HPC systems. It can help HPC users to accurately predict application performance in advance and determine optimum number of processors and wall time that they apply for. Then, their applications can run quickly and achieve good speedup without wasting resources. For HPC systems, accurate performance estimates provided by users can assist the system scheduler to decide efficient allocation and scheduling strategies, which can reduce idle waiting time and improve the HPC system utilization.

Acknowledgment

This work is supported by the National Key Research and Development Plan under grant No.2017YFB0202901, the Key Research and Development Program for Guangdong Province (2019B010136001), the National Natural Science Foundation of China(NSFC) under grant No.61672186.

References

[1] N. Attig, P. Gibbon, and T. Lippert, "Trends in supercomputing: The european path to exascale," *Computer Physics Communications*, vol. 182, no. 9, pp. 2041–2046, 2011.

[2] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*. ACM, 2011, p. 6.

[3] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 45.

[4] Y. Fan, P. Rich, W. E. Allcock, M. E. Papka, and Z. Lan, "Trade-off between prediction accuracy and underestimation rate in job runtime estimates," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 530–540.

[5] 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*. ACM, 2015, p. 64.

[6] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 368–377.

[7] J. Zhai, W. Chen, W. Zheng, and K. Li, "Performance prediction for large-scale parallel applications using representative replay," *IEEE Transactions on Computers*, no. 7, pp. 2184–2198, 2016.

[8] H. Casanova, A. Legrand, and M. Quinson, "Simgrid: A generic framework for large-scale distributed experiments," in *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*. IEEE, 2008, pp. 126–131.

[9] 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 ACM/IEEE Conference on Supercomputing*. IEEE, 2001, pp. 39–39.

[10] A. Hoisie, O. Lubeck, and H. Wasserman, "Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 330–346, 2000.

[11] C. Mei, "A preliminary investigation of emulating applications that use petabytes of memory on petascale machines," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2007.

[12] A. Bhattacharyya and T. Hoefler, "Pemogen: Automatic adaptive performance modeling during program runtime," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014, pp. 393–404.

[13] 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*. ACM, 2007, pp. 249–258.

[14] R. J. Hyndman and A. B. Koehler, "Another look at measures of forecast accuracy," *International journal of forecasting*, vol. 22, no. 4, pp. 679–688, 2006.

[15] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, pp. 1319–1360, 1994.

[16] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[17] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[18] J. Sun, S. Zhan, G. Sun, and Y. Chen, "Automated performance modeling based on runtime feature detection and machine learning," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. IEEE, 2017, pp. 744–751.

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

[20] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for gpu-based computing," in *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, vol. 1. Citeseer, 2009.

[21] D. H. Bailey, "Nas parallel benchmarks," *Encyclopedia of Parallel Computing*, pp. 1254–1259, 2011.

[22] R. Hornung, J. Keasler, and M. Gokhale, "Hydrodynamics challenge problem," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2011.

[23] G. Lu, W. Zhang, H. He, and L. T. Yang, "Performance modeling for mpi applications with low overhead fine-grained profiling," *Future Generation Computer Systems*, vol. 90, pp. 317–326, 2019.

[24] 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*, vol. 124, pp. 1–13, 2019.

[25] W. Zhang, A. M. Cheng, and J. Subhlok, "Dwarfcode: a performance prediction tool for parallel applications," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 495–507, 2016.