International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 08 Issue: 03 | Mar 2021 www.irjet.net p-ISSN: 2395-0072

# Automated Performance Modeling of HPC Applications Using Machine Learning

**Subin Babu[1], Ajeesh S.[2], Dr. Smita C Thomas[3]**

*[1]M Tech Student, APJ Abdul Kalam Technological University, Kerala, India*
*[2]Asst. Professor, Mount Zion College of Engineering, Kadammanitta, Kerala, India*
*[3]Assoc. Professor, Mount Zion College of Engineering, Kadammanitta, Kerala, India*

---

**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. 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.*

***Key Words***: **High-performance computing, Analytical Modeling, Replay-based Modeling, Model Transferring, Instrumentation.**

## 1. INTRODUCTION

High-performance computing (HPC) is the ability to process data and perform complex calculations at high speeds. To put it into perspective, a laptop or desktop with a 3 GHz processor can perform around 3 billion calculations per second. While that is much faster than any human can achieve, it pales in comparison to HPC solutions that can perform quadrillions of calculations per second.

One of the best-known types of HPC solutions is the supercomputer. A supercomputer contains thousands of compute nodes that work together to complete one or more tasks. This is called parallel processing. It's similar to having thousands of PCs networked together, combining compute power to complete tasks faster. It is through data that groundbreaking scientific discoveries are made, game-changing innovations are fueled, and quality of life is improved for billions of people around the globe. HPC is the foundation for scientific, industrial, and societal advancements.

As technologies like the Internet of Things (IoT), artificial intelligence (AI), and 3-D imaging evolve, the size and amount of data that organizations have to work with is growing exponentially. For many purposes, such as streaming a live sporting event, tracking a developing storm, testing new products, or analyzing stock trends, the ability to process data in real time is crucial. To keep a step ahead of the competition, organizations need lightning-fast, highly reliable IT infrastructure to process, store, and analyze massive amounts of data.

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 [8], [9]. Therefore, the utilization of the entire HPC system can be improved. Second, the model can offer insights about application behaviors [5], 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 [8], [9]. 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 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 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], [11]). Be- sides, 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 uses machine learning techniques to fit the mapping function between performance 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 [1], [2], some key features are dynamically decided at runtime. In this case, input parameters also cannot cover all performance features.

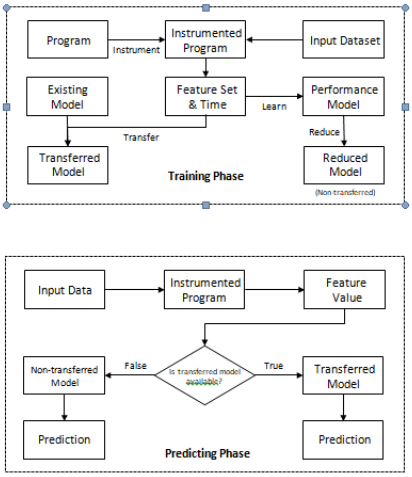


Fig. 1: Overview of automated performance modeling ofHPC applications using machine learning.

## 2. 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.

### 2.1 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 ex- ample, Eller et. al. [10] proposed an analytical model for Krylov Solver on structured grid problems. Hang et. al. [6] proposed a detailed performance model for deep neural networks. Barker's model [8] focused on the Krak Hydro-dynamics Application. In general, an analytical model for a specific application is difficult to be applied to other applications.

### 2.2 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 pre- diction 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 [4], [7] 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. [4] 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 bench- mark [7], storage requirements can range from hundreds of megabytes to tens of gigabytes for each run [9], [4]. 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.

### 2.3 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. [9] 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 [3], [5] 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. [11] 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], 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.

### 2.4 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. [2] 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. [1] used the Bayesian network to capture the parameter dependencies of an application. Maratheet. al. [3] 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.

The use of performance modeling manually has been explored before. There are approaches that focus on models generated for a very specific purpose but less on human readable general-purpose models. For example, Ipek and de Supinski propose multi-layer artificial neural networks to learn application performance [9] and Lee and Brooks compare different schemes for automated machine-based performance learning and prediction [1]. Zhai, Chen, and Zheng extrapolate single-node performance to complex parallel machines [2]. Wu and Muller extrapolate traces ¨ to larger process counts and can thus predict communication operations. Hoefler and Gropp. aimed to popularize performance modeling by defining a simple six-step process to create application performance models [3]. Bauer, Gottlieb, and Hoefler show how to model performance variations using simple statistical tools [4]. Another objective of performance modeling is to predict application performance on a different target architecture. Carrington et al. propose a model-based prediction framework for applications on different computers [5], Marin and Mellor-Crummey demonstrate how application models can be derived semi-automatically to predict performance on different architectures [6], and Yang, Ma, and Muller ¨ model application performance on different architectures by running kernels on the target architecture [7]. Besides program inputs, predictors generated from hardware features can be used in our hybrid approach to model performance of applications across architectures. This poses an interesting direction of future work. The authors of the Statistical Stall Breakdown [8] describe a mechanism that samples hardware counters and dynamically multiplexes hardware counters to compute a breakdown model for a PowerPC based microprocessor. The work by Huck et al. [9] focuses on automating the

process for parallel performance experimentation, analysis and problem diagnosis. Such mechanism is built on top of the PerfExplorer performance data mining system combined with the OpenUH [10] compiler infrastructure. The PerfExpert [3] tool employs the HPCToolkit [5] measurement system to execute a structured sequence of performance counter measurements to detect probable core, socket and node-level performance bottlenecks in important procedures and loops of an application. The work of Pavlovic et al. characterizes the memory behavior, including memory footprint, memory bandwidth and cache efficiency of several scientific applications. Based on the analysis of the executions of such applications they also estimate the impact of the memory system on the amount of the instruction stalls and on the real computation performance. Their results are shown per application execution, summing up all the information from the different tasks. There are other performance tools that exploit processor hardware counters and that have integrated sampling capabilities into their analyses. Tools like TAU, Scalasca [4], HPCToolkit, use sampling in addition to instrumentation, their sampling capabilities are mainly focused on assigning time consumption to source code lines instead of providing finer details on the hardware counters. Gonzalez, Gimenez and Labarta present a tool that automatically characterizes the different computation regions of the program [5]. Llort, Gonzalez and Servat detect clusters based on IPC and number of instructions committed and then detects the change of performance counters like cache misses inside the clusters [6]. Alam and Vetter propose code annotations, called "Modeling Assertions" [7] that combine empirical and analytical modeling techniques and help the developer to derive performance models for his code. Kerbyson and Alme propose a performance modeling approach [8] that is based on manually developed human expert knowledge about the application. Those modeling techniques rely on empirical execution of serial parts on the target architecture and are usually applied to stable codes which limits their usefulness during software development. The recent automatic online performance modeling strategy [5] had serious limitations as described in previous sections. Our hybrid strategy is the first technique that combines the knowledge obtained from static analysis and the power of dynamic analysis to produce more precise model.

## 3. EXISTING METHODS

Many researchers have previously been carried out in this field of phishing detection. Gathered the information from various such works and have profoundly reviewed them which has helped us in motivating our own methodologies in the process of making a more secure and accurate system. 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.

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.

### 3.1 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 [5]. 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 [8]. We describe the instrumentation of these different types of features, respectively, below.

#### 3.1.1 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.

### 3.1.2 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.

### 3.2 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 n samples. Each sample is expressed as (x, y), where x is a vector consisting of m features and y is the corresponding execution time. The goal of this regression problem is to find a mapping relation, f : x −→ y, that minimizes the mean square error (MSE) between the predictive value and the real execution time in the n samples:

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

There exist numerous approaches to solve this regression problem, like ridge regression [11], LASSO [4], artificial neural network (ANN), and random forest [12]. We adopt a random forest approach with an optimization called extremely randomized trees [2]. 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 [5]. 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.

## 4. PROPOSED SYSTEM

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

### 4.1 Experimental Setup

#### *4.1.1 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 P ROC, 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 [5]. The Intel MPI version 5.0 was used as the MPI library.

### 4.2 Feature Reduction

We first evaluated the feature reduction methods introduced in Section 3.4. 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 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, whensetting 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:

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

where ypredict[i] is the predicted performance and yreal[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 2.3 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 reduction may remove some important features and result in a higher error rate.

### 4.3 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], [11]. The other one is a deep learning model called PerfNet [2]. 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.

## 5. CONCLUSIONS

In this seminar, 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.

**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. Mladenic, and A. Skowron, edi-ˇtors, 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, J. Friedman, C. J. Stone, and R. A. Olshen. Classification and regression trees. CRC press, 1984.