Learning Cost-Effective Sampling Strategies for Empirical Performance Modeling

Marcus Ritter*, Alexandru Calotoiu*, Sebastian Rinke*
Thorsten Reimann*, Torsten Hoefler†, Felix Wolf*

*Technical University of Darmstadt, Department of Computer Science, Germany
†ETH Zürich, Department of Computer Science, Switzerland
{ritter,calotoiu,rinke,wolf}@cs.tu-darmstadt.de,
thorsten.reimann@sc.tu-darmstadt.de, htor@inf.ethz.ch

***Abstract*—Identifying scalability bottlenecks in parallel applications is a vital but also laborious and expensive task. Empirical performance models have proven to be helpful to find such limitations, though they require a set of experiments in order to gain valuable insights. Therefore, the experiment design determines the quality and cost of the models. Extra-P is an empirical modeling tool that uses small-scale experiments to assess the scalability of applications. Its current version requires an exponential number of experiments per model parameter. This makes the creation of empirical performance models very expensive, and in some situations even impractical. In this paper, we propose a novel parameter-value selection heuristic, which functions as a guideline for the experiment design, leveraging sparse performance-modeling, a technique that only needs a polynomial number of experiments per model parameter. Using synthetic analysis and data from three different case studies, we show that our solution reduces the average modeling costs by about 85% while retaining 92% of the model accuracy.**

***Index Terms*—Performance analysis, performance modeling, reinforcement learning, high-performance computing, parallel processing**

I. Introduction

The design and development of high-performance computing (HPC) applications is an extremely challenging task and requires major resource investments in order for them to run efficiently on existing and future large-scale machines. As the demand for parallelism and HPC is constantly increasing, so is the need for performance analysis. One way of analyzing the performance of a parallel program is to conduct experiments to explore its design and configuration space. However, often these spaces are very large, making a complete traversal, and therefore the determination of the performance, economically infeasible. Performance modeling offers another, more promising way by providing analytical expressions of the application behavior at larger scales. One example for a performance model is the expression of the execution time as a function of the number of processes and the problem size. Analytical modeling is one way to derive such models, by experts inferring application behavior from code analysis. Therefore, they are expensive and usually limited to a few selected program kernels [1], whereas empirical models can be generated automatically based on only a few small-scale experiments [2]. Apart from their cost efficiency and increased code coverage, they have proven to be helpful to find scalability bugs [2] and accurately predict the performance of parallel programs [3]. However, with an increasing number of configuration parameters even the experiments needed for empirical performance modeling become very expensive.

Extra-P [4] is an empirical modeling tool that uses small-scale experiments to assess the scalability of an application. Its current modeling approach requires experiments of all combinations of the considered application parameters [5]. Additionally, it suggests five repetitions per experiment to counter the effects of system noise, as well as five different values for each parameter considered [5]. Thus, the number of experiments e it needs grows exponentially with the number of parameters m, amounting to $e = 5^{(m+1)}$. Consequently, Extra-P needs 625 experiments in order to model an application with three varying configuration parameters, which is usually too expensive to be practical.

In this work, we propose a novel sparse performance-modeling technique, which only needs a polynomial number of experiments per model parameter. Furthermore, we investigate the trade-off between experiment repetitions to reduce the effects of noise and conducting new experiments to capture potentially unseen behavior. In addition, our new modeling approach also offers more freedom for the experiment design, as we do not need to analyze all parameter combinations anymore. Motivated by the newly attained flexibility we use reinforcement learning (RL) to train an agent on the task of parameter-value selection. We then use the agent acquired knowledge to derive a heuristic selection strategy, which functions as a design guideline for the experiments required by our approach, and is generally valid for all kinds of HPC applications. Depending on factors such as the level of noise or the number of model parameters, the heuristic determines

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. The work has been supported in part by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH16008D, and by the German Research Foundation (DFG) through the Program "Performance Engineering for Scientific Software" (BI 714/6-1), and the *ExtraPeak* project, and by the US Department of Energy under Grant No. DE-SC0015524. Calculations were performed on the Lichtenberg computing cluster at TU Darmstadt, the Vulcan Supercomputer at Lawrence Livermore National Laboratory, and the SuperMUC Petascale System at Leibniz Supercomputing Centre.

which experiments should be conducted in order to reach an optimal trade-off between model accuracy and cost reduction. In combination, our solution allows us to create cheap empirical performance models of large-scale parallel applications with multiple configuration parameters such as FASTEST [6], Kripke [7], and Relearn [8]. The major contributions of our work are:

- A sparse performance-modeling technique that requires only a polynomial number of experiments per model parameter and allows an overall more flexible experiment design.
- A parameter-value selection heuristic that leverages the sparse performance-modeling and functions as a design guideline for the required small-scale experiments, reducing the average modeling costs by about 85% while retaining 92% of the model accuracy.
- Three case studies that demonstrate the advantages and the substantial cost reduction our solution can achieve as well as its inherent limitations.

The remainder of the paper is organized as follows. After providing background knowledge on our prior work related to automated empirical performance modeling in Section II, we outline our novel approach in Section III. In Section IV, we evaluate the sparse modeling technique using synthetic analysis and compare accuracy and cost with our previous work. We then present three case studies and discuss the insights in Section V. Finally, we take a look at related work in Section VI and provide a conclusion in Section VII.

II. Background

Our approach builds upon Extra-P [2], a tool that leverages empirical measurements to create performance models. The input of Extra-P is a set of measurements representing change in metrics such as runtime or floating point operations as configuration parameters such as the number of processes or the problem size per process are varied. The output of Extra-P is a human readable function showing how the metric of interest changes as the parameters change, similar to the complexity of an algorithm. One example for such a function is the performance model we created for the scientific application FASTEST [6]. For one of its main kernels `celuvw` we modeled the runtime T in seconds, where $T(p,s) \approx s^{7/4} + p^{4/3} \cdot \log_2(p) \cdot s^{7/4}$ is a function of the number of processes p and the problem size s.

The approach is based on the observation that the complexity of algorithms used in both sequential and parallel applications is usually a combination of polynomial and logarithmic expressions. The performance model normal form (PMNF) codifies this insight and expresses the effect of a number of parameters x_i on a metric as a sum of terms consisting of products of polynomial and logarithmic expressions in the parameters x_i. The expression is formalized in Equation 1.

$$f(x_1,\ldots,x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \qquad (1)$$

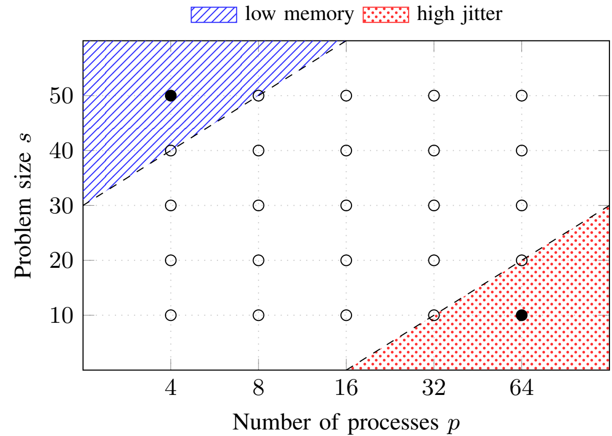


Fig. 1: Example for a set of performance measurements used for 2-parameter analysis. The filled circles represent the extreme measurement cases, namely those where the most work is divided among the fewest processes and where the least work is divided among the most processes. Both cases are inherently difficult to measure, as the former often runs for longer than system limits allow or does not fit the available memory, while the latter has so little work per process that the dominant effect measured is the system jitter.

To generate models, a search space of possible model hypotheses is automatically generated by instantiating Equation 1 with different sets of exponents. To select the best models out of the set of hypotheses, we first compute the coefficients for each hypothesis using regression and subsequently evaluate it using cross-validation. A new search space is generated in the vicinity of the best model found and the process is repeated iteratively until no significant improvements to the models can be made [9].

Currently, Extra-P first models the effect of each parameter x_i on performance separately. The developers of Extra-P suggest five different values for each parameter are required to obtain accurate models if at most one non-constant term is permitted, while the values of all other parameters are fixed, and they suggest each measurement be repeated five times to alleviate the impact of noise [10]. The core heuristic of the multi-parameter modeling approach is the assumption that the best multi-parameter model can be derived by combining the best single-parameter models in different ways and selecting the one that best fits the data. An important optimization comes in at this point, allowing the modeling process to eliminate parameters that do not have an impact on performance. After determining the single-parameter models, all possible additive and multiplicative combinations of these selected single-parameter models are tested to determine the multi-parameter model that fits all measurements best.

The current implementation requires measurements representing all combinations of all parameter values to provide correct results. This requires an exponential increase in measurements as the number of parameters to be considered

grows. Given the five measurements per parameter required and the five repetitions of each measurement suggested, the cost incurred is 5^{m+1} measurements, where m is the number of parameters considered. Apart from the sheer number of measurements required, which is daunting enough by itself, it is sometimes difficult to generate a full set of measurements where for each parameter value all combinations of all other parameter values are available. For example, when considering problem size and number of processes, it is difficult to find a range for these parameters such that the smallest problem size still offers enough work such that the largest number of processes considered can be used efficiently, and at the same time the largest problem still fits in the memory of the smallest number of processes considered and can be computed without reaching the limits given by the job-system where the experiment is being performed. Figure 1 shows both a matrix of the 25 measurements required to model a common case, the effect of problem size and process count on performance, and highlights the extreme cases previously discussed.

The reason such a large number of measurements is necessary can be found in the first step of the modeling process, where the single-parameter models are created. The values for this step are created by averaging all available measurements for each distinct value of a parameter across all measurements where the parameter of interest has the particular value. Considering each parameter as an independent variable in an n-dimensional space, this is in effect a projection from the vector space of all parameters into the vector space of just one parameter. For this projection to only show the effect of the focused parameter, the impact of all other parameters on each value must be the same. This can be achieved by ensuring that the exact same combination of values for other parameters are available.

III. Approach

In this section, we first formulate the basic idea of generating performance models from a sparse data set and then explain how we train a reinforcement learning agent to teach us a practical parameter-value selection strategy. In contrast to the previous approach discussed in Section II, which the current version of Extra-P is based upon, the number of experiments required for sparse modeling grows only polynomially (i.e., almost linearly) with the number of model parameters and makes the experiment design more flexible. The reinforcement learning agent exploits this newly attained freedom in the experiment design to find an strategy that represents a near-optimal trade-off between retaining model accuracy and reducing the overall modeling cost.

A. From exponential to linear cost

We have made the observation that the previous (i.e., dense) method assumes that there is one and only one behavior with respect to each parameter across the entire measured space. If this is true, the same function terms describing the effect of a given parameter should be identified no matter which sequence of five measurements is considered as long as the effects of all other parameters are kept constant. Rather than requiring every combination of values selected for each parameter, it could be sufficient to select a sequence of five measurements for each parameter to create single-parameter models. However, a thorough analysis is required to ensure that lowering the number and cost of measurements is not detrimental to the quality of the results, given the empirical nature of the modeling process. Figure 2 shows how only a subset of measurements can be sufficient to generate a single-parameter model that describes the impact of the number of processes on performance.

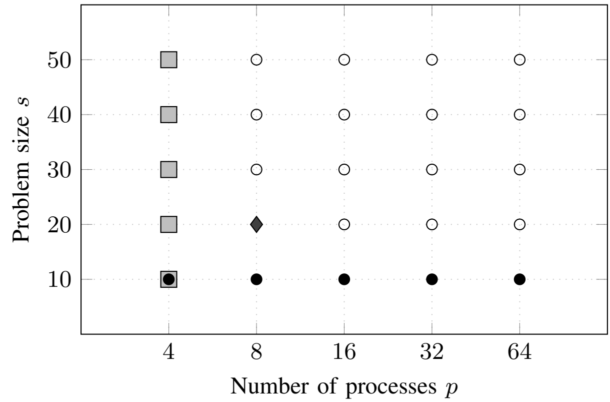


Fig. 2: Performance measurement set example for 1 and 2-parameter analysis. The filled black circles represent a set of measurements that would be sufficient to generate a single parameter model for how the number of processes affects performance. Combined with the filled gray squares they are an example of a subset of points that is sufficient to create a performance model for how p and s affect performance. The filled diamond represents a potential additional measurement point that could be used to increase model accuracy.

Extra-P allows the effect of different parameters on a metric to be either additive, meaning that the effect of those parameters is independent of each other, or multiplicative, meaning that their effect is compounded. While more interactions are theoretically possible, they have not been discovered in practical codes. The binary decision whether the effect of any parameter pair is additive or multiplicative cannot, however, be made with only a sequence of five measurements for each parameter. At least one additional data point is required, one that is not part of those sequences. We must analyze if adding more data points provides additional accuracy and discover whether a sweet spot exists where the quality is close to optimal but uses significantly less or cheaper measurements. In our synthetic evaluation, the addition of this extra point improves the model accuracy from 61.8% to 71.8%, whereas adding more data points delivers only diminishing returns. More details will be presented in Section IV.

Figure 2 shows a set of measurements which are sufficient to correctly identify two-parameter performance models. Of course, any of the columns and rows could be used to generate the performance model. They can even overlap with each

other, so that a measurement point is used for modeling the effects of several and not just one parameter. The question of which rows and columns to measure as well as which additional points to consider such that the best models can be generated at the smallest cost is still open. While using the cheapest set of measurements required to generate a model at all is appealing, we must quantify how the quality of the models degrades compared to other strategies. A large concern is that even if the assumption holds that only one behavior exists across the entire domain, the effects of jitter on those measurements, where less actual work is performed, is stronger, and could potentially jeopardize the accuracy of the resulting models. Furthermore, if the measurements contain outliers, these will more significantly affect the results if fewer measurements are used.

In order to answer these questions we use reinforcement learning and train an agent on the task of parameter-value selection. More specifically, we want the agent to find an efficient strategy of selecting the measurement points that represents the optimal trade-off between retaining model accuracy and reducing the overall modeling cost. Now, after having specified the task the agent has to learn, we define the environment it will be exposed to. We use this environment to both train the agent and to evaluate different scenarios. Then, we introduce the RL agent, including the setup of the neural network as well as the RL cycle, followed by a detailed explanation of the agent's training and validation process. Finally, we use the acquired knowledge to derive the selection heuristic.

B. Evaluation environment

For each evaluation we generate test functions by instantiating our performance model normal form from Equation 1 with random coefficients $c_l \in (0.01, 1000)$ and random exponents i_l and j_l, selected from the sets $I = \{0, 1, 2, 3\}$ and $J = \{0, 1, 2\}$. We then evaluate each function in our test set for 5^m parameter-value combinations, where m is the number of parameters. Because of limited computing resources, we used three parameters for training, but evaluate the resulting heuristic later with up to four. Depending on m, the parameter values are drawn from predefined sets: $x_1 \in \{4, 8, 16, 32, 64\}$, $x_2 \in \{10, 20, 30, 40, 50\}$, $x_3 \in \{1000, 2000, 3000, 4000, 5000\}$ and $x_4 \in \{2, 4, 6, 8, 10\}$. x_1 represents the number of processes p and has a special meaning when determining the cost of experiments.

To simulate the noise one would experience on a production system, we apply different levels of random noise $\in \{0, \pm1, \pm2, \pm5\}$ % to each evaluation. For each measurement configuration and noise level, we draw up to seven samples. This provides us with a set of $rep * 5^m$ measurements, where rep is the number of samples taken of each measurement configuration, and m is the number of parameters considered. We use these measurements as input for our two model generators and to train our reinforcement learning agent. While the previous modeler implementation requires all of these measurements for modeling, the sparse modeler only uses a small subset of points, depending on the agent's strategy.

To evaluate the results, we consider two aspects, accuracy and cost. To define cost, we try to replicate the scenario of the tool being used to determine the scalability of an application, which is the one most often encountered in practice. As mentioned before, we imagine that one parameter, x_1, represents the number of processes p and that the metric we measure is the total runtime per process. The cost of the measurement (e.g., the total core hours to run the experiment) is therefore not the same for all samples. Given that the exhaustive modeler requires all measurements, we consider their accumulated cost to be 100%, and determine which percentage of it the sparse modeler requires. To define accuracy, we consider the next parameter value in every series (e.g., $x_1 = p = 128$, $x_2 = 60$, $x_3 = 6000$, and $x_4 = 12$) and evaluate the resulting model for this combination. Then, we verify whether the predicted value is within $\{\pm5, \pm10, \pm15, \pm20\}$ % of the actual value.

For the sake of completeness, we have also investigated the effect of adding more measurements for each parameter, using six or seven samples rather than five. We observed that the optimum in terms of cost and benefit depends on the number of parameters considered. Specifically, we found that repeating each measurement four times provides the optimal cost/benefit ratio if two or three parameters are considered, while for single-parameter models even two repetitions can be sufficient. When considering four or more parameters, six or more repetitions improve the accuracy significantly.

C. Reinforcement learning agent

We instantiate the agent for the three configuration parameters used for training. Using five different values per parameter and all of their combinations provides us with a matrix of 125 possible measurement points.

The first step towards solving this optimization problem with RL is to describe the task of parameter-value selection as a Markov decision process. For this purpose, we define a set of environment states S, a set of actions A, and a reinforcement signal R.

Each state s is represented by a list of values, containing four major elements $s[p, i, j, c]$: a one-dimensional representation of the matrix of measurement points p, the selection phase i, the selection step j, and the model cost c. The matrix representation itself is a list of 125 binary values $[1, 0, 1, ...]$, with a '1' denoting a point that is used for modeling and a '0' the opposite. The selection phase i is a single-digit integer number with a value of either [0, 1, 2, or 3] which instructs the agent whether it needs to select a line of five measurement points for parameter one, two, or three or whether it needs to select additional individual points to further increase the model accuracy. We introduced this simplification to drastically reduce the number of point combinations the agent can produce. Moreover, it supports the restriction that our sparse modeling approach cannot work with an arbitrary set of points. The step indicator j counts the number of points the agent has already selected. Finally, the model cost c accumulates the relative cost of these measurement points in percent in comparison to the complete matrix.

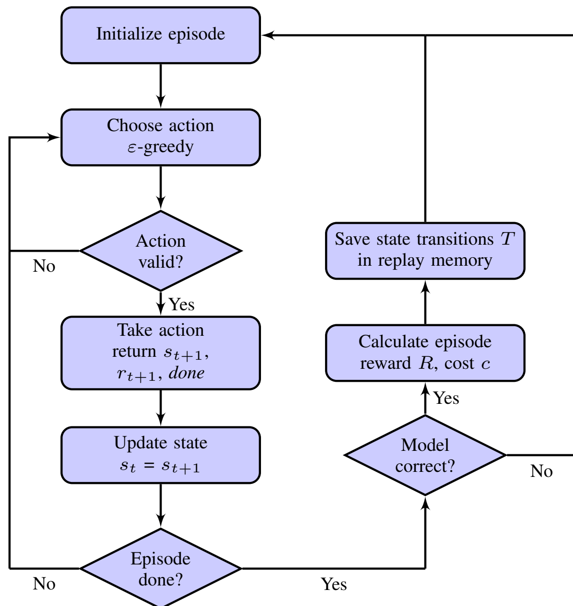


Fig. 3: Overview of the agent's reinforcement learning cycle. One traversal of the flowchart euqals one episode of training.

The set of possible actions is determined by the set of measurement points that can be selected by the agent. For each point we define an action a that allows the agent to select that point. Furthermore, we have one additional action per parameter, which allows the agent to select a line of five points in a row at a time.

The reinforcement signal $R_a(s_t, s_{t+1})$ is the immediate reward the agent receives after taking an action a to move from state s_t to s_{t+1}. We calculate the reward as a function of the cost c of the selected points p where $R_a(s_t, s_{t+1}) = 1 - c$ in the interval $(0, 1)$.

Based on these definitions, we use a modified version of the standard RL cycle [11] and Q-learning [12] as the algorithm to search for an optimal policy for the selection of actions (i.e., measurement points). Our RL agent is based on the DQN approach by Mnih et al. [13], although we use a simpler multilayer perceptron consisting of an input layer, two fully connected hidden layers, and one output layer with softmax. Nevertheless, the neural network is necessary as we have to deal with a very large number of agent states and its approximation of the state space improves the training performance. This network architecture allows us to feed the agent's state transitions $T[s_t, a_t, r_{t+1}, s_{t+1}]$ directly into the network and yield the Q-value for each action. Furthermore, we use several additional optimization techniques, such as prioritized experience replay [14], double Q-learning [15], and double DQN [16], to further improve the learning performance.

Figure 3 outlines the basic RL cycle of our agent that is embedded in the training process. For each training episode, we first generate a random performance function and create the initial state s_I. Then at each time step t, the agent will choose an action a_t using an ε-greedy strategy. After a validation of the chosen action (we do not allow the same sample to be taken repeatedly), the agent will either choose another action or execute the chosen action in the environment. In the latter case, the agent and the environment move one step further in time, from t to t_{+1}. In return, the agent will receive the new state s_{t+1}, the reward r_{t+1} associated with this transition, and a boolean value that determines whether the episode is finished. We then backup the old state s_t before updating it to s_{t+1}. This process is repeated until the episode is finished.

The termination of a training episode can have several reasons. The simplest one is that the agent has predicted the correct function. Other possible reasons are that all available measurement points have been selected without predicting the correct function or that the agent reaches a pre-defined number of maximum steps it is allowed to take. Such a limit can be useful in order to speed up the training process, especially when the agent is exposed to high degrees of noise.

After the episode has been terminated, we check the accuracy of the predicted model by evaluating the scaling properties, as explained in Section III-B. If the model is accurate enough, we accept it as correct and calculate the overall cumulated reward R and cost c of the episode. Finally, we save all state transitions T the agent has experienced to a replay memory, as we do not directly update the neural network during each training episode. Moreover, if the predicted model is too inaccurate we discard all knowledge of the episode. There is no sense in training the agent on examples that can not be solved by our modeling approach due to the impact of noise—no matter which measurement points it selects. This concludes the basic RL cycle of parameter-value selection.

D. *Agent training and validation*

We trained our parameter-value selection agent considering three configuration parameters, over a set of 200,000 distinct performance functions (200 epochs with 1,000 episodes each), using the evaluation environment previously described. During the first training epoch, we let the agent choose its actions completely at random, without updating the neural network. We save the state transitions the agent experiences to the replay memory until we successfully predicted 1,000 models. Starting with the second epoch, we use an ε-greedy strategy to choose our actions. We save the new state transitions to the replay memory, and perform a learning update after each episode, allowing the agent to train on specifically selected transitions from the replay memory. Until we reach 50 epochs, we anneal ε linearly from $[1.0, 0.1]$, reducing the amount of exploration in order to exploit the already acquired knowledge. At the end of each training epoch, we compare the current network with the best network previously found, by testing over 100 different functions which were not part of the training set and determining the network that reaches the highest overall reward. The winner becomes the new best network and will be used as a baseline for training in the next epoch.

Figure 4 shows the progress of the training process, the average cumulated reward and modeling cost in percent the agent achieved after each training epoch. The agent starts with

an average reward of 0.88 which in turn means a cost of $\approx$ 12% using a blank network, choosing actions completely at random. Therefore, the environment restrictions dictated by the selection phase, such as forcing the agent to select five points in a row for each parameter, helps the agent to achieve good results even without having acquired a lot of prior knowledge. Over the next 50 epochs, we see that the average model cost converges to about 4.5% and the reward is constantly rising. Beyond this point, the agent is no longer able to further increase the reward (i.e., produce the same number of correct models at lower cost), instead the reward oscillates around 0.95. Experiments with other configurations of the training process show similar results. The agent finishes the training process with an average reward of 0.95, and an average percentage cost of 5.53%, performing $\approx$ 3.1 steps (i.e., selecting a line of 5 points for the first parameter, a further line of 4 points for the second parameter and finally 4 points for the third parameter. By having a common point across all parameters we can create single-parameter models with just 13 points). Most of the models can be predicted correctly using only the 13 base points, although some models require additional points, in extreme cases even up to 100 out of 125. These cases slow down the training process and prevent the agent from optimizing the models we actually want to optimize, which are the ones that can be predicted using less measurements. If almost all points are required to correctly predict a model, there is not much room for optimization. Therefore, we terminate these episodes after the agent exceeds a predefined maximum number of steps.

We validated the agent using the same procedure as for training, only without updating the network. In order to reduce the chance of overfitting our training data, we let the agent choose random actions with a probability of 5%. In addition, we use a fixed set of 100,000 evaluation functions the agent has not seen before and apply $\pm 5\%$ noise to the measurements. On average, our agent achieves a reward of 0.96 and uses 3.85% of the cost. Therefore, it chooses 14 points on average. These results represent the most efficient strategy the agent could learn, that equally optimizes model cost and accuracy for all training functions.

E. Selection heuristic

The agent is able to correctly predict all of our 100,000 test functions by using only 3.85% of the cost the previous version of the modeler would have required, proving that it is a generally valid strategy to start modeling by choosing the cheapest lines of five measurement points per parameter. Furthermore, it discovered that the sparse modeler most certainly requires more than these lines, to provide a correct prediction, although how many of these additional points are required depends on the function. The best strategy is to add the cheapest points available that have so far not been selected. For the vast majority of the functions we tested, 1 to 10 additional measurement points are enough to provide an accurate model. There are some functions, however, that require almost all points in order to provide a correct model, also depending

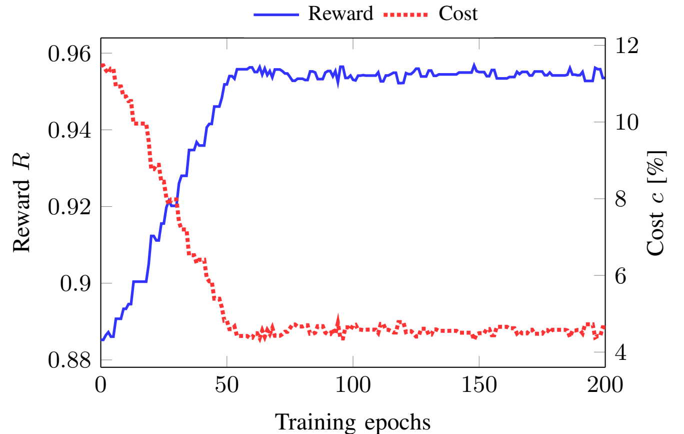


Fig. 4: Training curves illustrating the agent's learning progress by visualizing the average cumulated reward R and the relative modeling cost c in percent per training epoch.

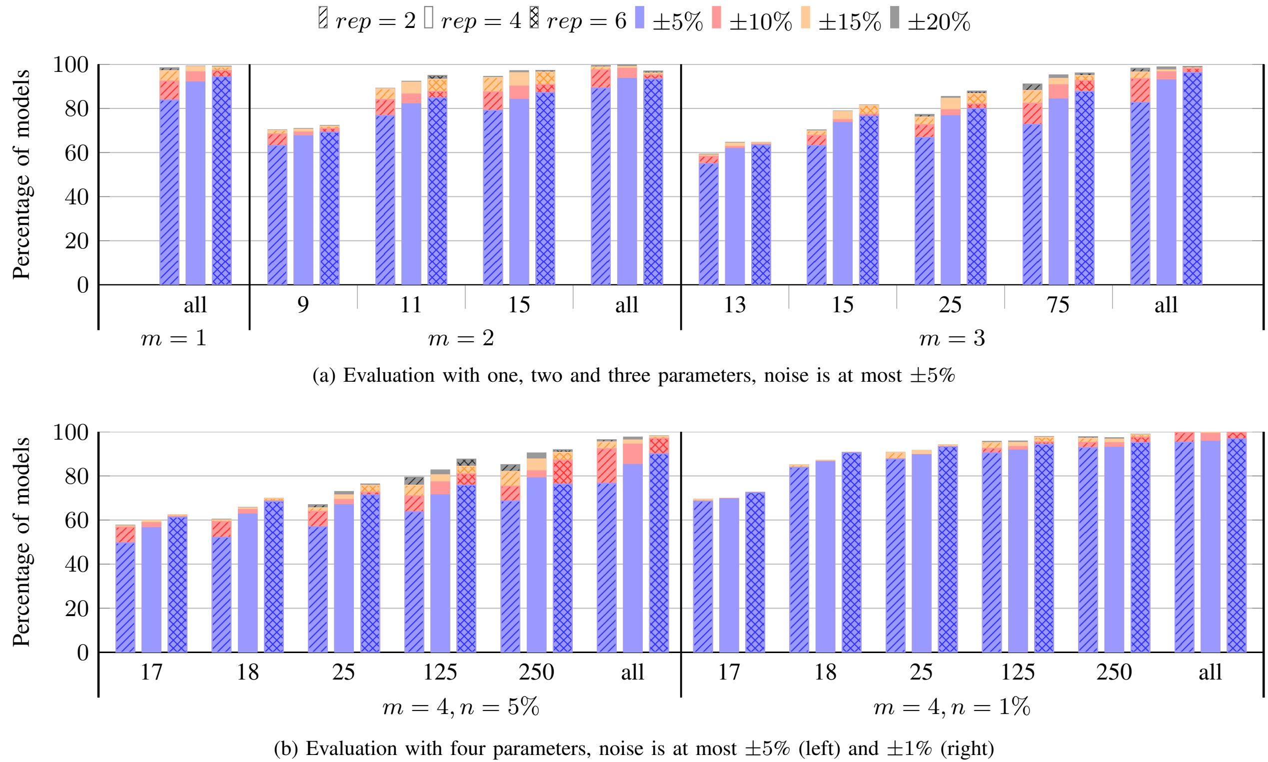
on the noise present in the measurements. Higher amounts of noise require more additional points, and benefit from having more repetitions for each measurement point. In general, we can formulate the following heuristic:

1) Measure the cheapest available lines of five points per parameter. Use these to create a first model using our sparse modeling technique.
2) Perform an additional measurement, starting from the cheapest ones available. Using the model previously determined one can assess if the quality of the model is sufficient or additional points are required.
3) Recreate the model using all available points.
4) If the quality of the model evaluated in step 2 is insufficient, return to step 2.

Using this strategy, one can increase the accuracy of the model until the budget available for experiments is exhausted. Although our agent was trained with only three parameters, we will show in Section IV that it can be effectively used even for performance models with four parameters.

IV. Evaluation

To evaluate the accuracy and cost effectiveness of the sparse modeling technique in comparison with our previous approach, we conducted an extensive synthetic data analysis using the described selection strategy. We experimented with different modeler configurations and parameter-value selection strategies for up to four configuration parameters. Figure 5 provides an overview of the experiments we conducted using the same evaluation environment and definitions of accuracy and cost, as for the RL agent, that we outlined in Section III-B. As already explained there with an example, we declare a model as accurate if the value it predicts for a selected parameter value combination beyond the range used to derive the model is within $\pm 5\%$, $\pm 10\%$,$\pm 15\%$, or $\pm 20\%$ of that expected value. The value combination is derived by choosing for each parameter the first value outside this range. We



(a) Evaluation with one, two and three parameters, noise is at most ±5%

(b) Evaluation with four parameters, noise is at most ±5% (left) and ±1% (right)

Fig. 5: Comparison of the accuracy of our previous and the sparse modeling approach for different numbers of parameters $m = \{1, 2, 3, 4\}$ and measurement repetitions $rep = \{2, 4, 6\}$ over 100.000 synthetic performance functions. The x-axis describes the number of points that is used for modeling. For each configuration we outline the percentage of models where the prediction is within ±5%,±10%,±15% and ±20%.

define cost as the percentage of compute time used to derive the model compared to the accumulated compute cost of all measurements required by the old modeling approach. We use the value of the function to be modeled as cost, therefore not all measurements will contribute the same value. To ensure the transparency of our method and our results, the reader may access an extended version of the evaluation at https://github.com/anonymous117/paper.

a) One parameter: For one parameter, there is no difference between the two modeling approaches, and given that only one parameter must be modeled, the effect of noise will usually be too small to affect results. The results are very accurate: considering four samples for each measurement ($rep = 4$) over 92% of models are within ±5% of the actual values and almost 97% are within ±10%. We believe even two samples ($rep = 2$) should be sufficient for single parameter modeling on most systems, as ≈ 86% of results are within ±5% of the actual values, and over 92% are within ±10%.

b) Two parameters: The strategy of the sparse modeler for the selection of parameter values in this case is quite simple, suggesting the cheapest points will provide good results. The effect of ±5% noise on accuracy is such that sometimes the influence of one of the parameters cannot be correctly characterized by either modeling approach (i.e., the model only describes the effects of one parameter). Nevertheless, the results are satisfactory: considering four samples for each measurement ($rep = 4$), more than 93% of models are within ±5% of the actual values if all measurements are used and over 98% are within ±10%. The cost benefits of the sparse modeling approach are clear even for two parameters, but the actual reduction depends on the number of measurements used. Using 11 out of the 25 possible measurements, each repeated four times, we achieve 82% models within ±5% of the actual values, and 86.5% are within ±10%, while the cost is only 12.7% of the total.

c) Three parameters: The agent taught us that, while more expensive measurements provide better results in some cases, these are exceptions and most functions are correctly modeled using the cheapest points available. Additional points provide better results, but—similar to repetitions—they offer only diminishing returns. The effect of ±5% noise on accuracy is so great that it will often be comparable to the impact of at least one of the parameters, making accurate modeling very difficult. Using all measurements and four repetitions results in 93% of models within ±5% of the actual values. Using 15 out of 125 measurements, we still find almost 74% of models within ±5% of the actual values. If we increase the set of used measurements to 25, we increase the rate to 77% within ±5% and 85% within ±15%. The cost of gathering the measurements plummets, however, to only 1.8% in the

first case and 2.4% in the second.

d) Four parameters: When considering four or more parameters simultaneously, the impact of noise becomes more pronounced. We conducted experiments with measurements where noise is at most $\pm 5\%$ of the value of the measurement. Let us consider the best possible scenario: all parameters have equal contributions to the measurements, making each of them equally challenging to model. The contribution of at least one of the parameters can be at most 25% of the value of a measurement if four parameters are considered. This in turn means the impact of noise is almost half the impact of a parameter of interest, making it quite likely that the resulting model will not be entirely correct. While the parameters with most impact can be modeled correctly, those with a smaller contribution can only be reliably modeled if the impact of noise is kept much smaller than that of relevant parameters. Therefore, we suggest additional repetitions if more than three parameters are considered when modeling runtime. However, there is a range of useful hardware and software counters, such as the number of floating-point operations performed or the number of bytes send or received over the network, which are inherently far less sensitive to noise because they simply count the work performed—no matter how long it takes—and are therefore fairly reproducible—even on a noisy system. A more detailed discussion regarding noise-resilient performance counters can be found in Calotoiu et al [10]. For such metrics, functions with more parameters can be accurately modeled even without adding more points or repetitions. In any case, even if the resulting models are not perfect, and not all parameters are correctly identified, the model can still provide useful insights: arguably, if the contribution of a parameter is smaller or comparable to the impact of noise there is little value to be gained from modeling it at all.

As shown in Figure 5b, we consider two, four, and six repetitions for each measurement. We observe that even when we use all possible measurements, only 76% of models are accurate within $\pm 5\%$ of the expected value if only two repetitions are considered. This number grows to 90% if six repetitions are considered. The value of repeated measurements for the overall result quality is higher the more parameters are considered. Furthermore, the decrease in accuracy is due to the noise preventing the modeler from capturing the effect of parameters with smaller contributions: There are many models which are not accurate within $\pm 5\%$, but are accurate within $\pm 20\%$ of the expected value (depicted in the figure by the darker rectangles at the top of the bars). These appear significantly more frequently for four parameters than for two and three parameters.

The minimum number of points needed to model four parameters is 17. Looking at models accurate within $\pm 20\%$ of the expected value, even with so few points and six samples per point ($rep = 6$), we still get around 60% accuracy. Increasing the points to 25, which is still less than 0.45% of the cost of all 625 points, we advance to around 71% accuracy. Using 250 points, but only 4.74% of the total cost, we reach about 90% accuracy for four parameters. In order to get a better understanding of the relationship between the number of measurements and the model accuracy, as well as model cost, we used the results of the different modeler configurations as input to our modeling tool. Based on the evaluation data we can model the accuracy and cost of the sparse modeler as a function of the considered measurements. However, these functions are only applicable in case the measurements are sorted by their cost from cheapest to most expensive. The function for the cost is $c(x) = 0.282 + 4.80 \cdot 10^{-7} \cdot x^3$ and the function for the accuracy is $a(x) = 57.329 + 3.62 \cdot x^{1/3}$ where x is the number of measurements considered. Given that the cost of measurements is not constant, and measurements with larger problem sizes and more processes can be orders of magnitude more expensive than smaller tests, it will often by more effective to repeat smaller measurements than add more expensive measurements.

We therefore show that even in the presence of noise our models are capable of capturing the most relevant behavior, and guide user decisions. To showcase the usefulness of our method performs if the data available is less impacted by noise, we also performed experiments with noise being at most $\pm 1\%$ of the value of the measurements. This is qualitatively different from no noise, as in the presence of no noise the exact model can easily be found using very few measurements. However, even noise-resilient measurements such as the number of floating-point operations performed or the number of bytes sent or received over the network often show minute run-to-run variations. For example, if an operating-system daemon causes some floating point operations to be performed on its behalf, this could be conflated with the operations performed by the application and logged by the instrumentation system. While this disturbance is usually many orders of magnitude smaller than the floating-point operations performed by a numerical application, it can still lead to tiny variations in the results. We believe a noise of $\pm 1\%$ simulates a very stable metric with little to no run-to-run differences without relying on the assumption that there is absolutely no noise, which is very hard to conclusively prove when gathering measurements on realistic systems.

When allowing at most $\pm 1\%$ noise, we see in Figure 5b that the accuracy starts at almost 69% for two repetitions and 17 points and rises to over 95% for six repetitions and 250 out of a possible 625 points. As it is to be expected, both providing more measurements and increasing the number of repetitions from two to four to six improve accuracy, but there is much less benefit in adding more repetitions. Given that there is much less run-to-run variation, this is no surprise. We also see a much steeper increase in accuracy as more measurements are added, reaching over 87% with just 25 points and two repetitions. When the impact of noise is low, it is not only possible to obtain better models, but it is also significantly cheaper to do so.

e) Discussion: To summarize, it is possible to reduce the costs of modeling by one or even two degrees of magnitude while only giving up single-digit percentages of accurate models. Of course, using more points or more repetitions

improves results, but our recommendation is to always start from the cheapest set of measurements available and repeat each sample four times, and add additional measurements only if the quality of the results warrants it. The impact of noise is increasingly important as more parameters are considered, requiring the user to provide more repetitions and additional measurements if precise results are important. However, the general trend can still be observed with very little investment.

V. Case Studies

Below we present three case studies that demonstrate the advantages and the substantial cost reduction our solution can achieve as well as its inherent limitations. For each of them we create multi-parameter performance models of their main kernels using the sparse modeling technique and the described parameter-value selection heuristic. We discuss the insights the models provide and analyze their accuracy based on an evaluation of their scalability. Therefore, we follow the same approach as defined in Section III-B. However, as we are working with real application data, we are trying to analyze the measurement point which is the furthest away from the ones used for modeling.

A. FASTEST

The software package FASTEST [6] is a tool for the simulation of flows in complex three dimensional configurations. We measured its performance on SuperMUC, a petascale system at Leibniz Supercomputing Centre with more than 241,000 cores and a combined peak performance of about 6.8 petaflop/s. Using Score-P [17] we were able to acquire different metrics such as execution time and the number of floating point instructions. The analysis covers two configuration parameters: the number of processes p and the problem size per process s. To obtain the set of measurements required by our approach we varied the number of processes (16, 32, 64, 128, 256, 512, 1,024, 2,048, and 4,096) and the problem size (131,072, 65,536, 32,768, 16,384, and 8,192). We ran 125 tests (25 different parameter settings times 5 repetitions). In contrast to the square matrix of performance measurements shown in Figure 2 the resulting matrix for FASTEST is arranged in a parallelogram. As described in Section II, it is often difficult to generate a full set of measurements where for each parameter value all combinations of all other parameter values are available. In the case of FASTEST this is not possible, as we either run out of memory or the runtime is too small compared to the effects of jitter. Therefore, our previous approach would not have been able to model this application at all.

Following our heuristic strategy, we start by using only the minimum number of measurement points required by the sparse modeler, namely the two cheapest lines of five points to model the effects of p and s. Based on these nine points we automatically create empirical performance models for each kernel of FASTEST. The kernels we found most significant, and confirmed by the developer to be indeed critical to performance are: `celuvw`, `caluvw_sipsol`, `celp2` and `calcp_sipsol`. Table I shows the corresponding models. This information allows us to predict the scalability of FASTEST for specific configurations. For example we predict the runtime of `caluvw_sipsol` for the configuration $p = 512$ and $s = 65,536$, which we did not use for modeling. Analyzing FASTEST at a larger scale is extremely difficult, as we can conduct only a very limited amount of measurements for larger numbers of processes, due to the nature of its configuration. The actual measured average runtime value at this point is 669.49 seconds. With our performance model we predict an average value of 683.96 seconds, that is an error of only about 2%. Consequently, we can also use our models to identify scalability bottlenecks.

TABLE I: Selected performance models for different kernels of FASTEST, Kripke and Relearn describing their runtime T in seconds as a function of their configuration parameters.

Kernel	Model	$\hat{R}^2$
FASTEST		
celuvw	$s^{7/4} + p^{4/3} \cdot \log_2(p) \cdot s^{7/4}$	0.964
caluvw_sipsol	$s \cdot \log_2^2(s) + p^{1/2} \cdot s \cdot \log_2^2(s)$	0.989
celp2	$s^{4/3} + p^{4/3} \cdot \log_2(p) \cdot s^{4/3}$	0.939
calcp_sipsol	$s \cdot \log_2^2(s) + p^{1/2} \cdot s \cdot \log_2^2(s)$	0.984
Kripke		
LTimes	$d \cdot g^{5/4} \cdot \log_2(g) + d$	0.974
LPlusTimes	$d^{3/4} \cdot \log_2(d) + g^{3/2} \cdot \log_2(g)$	0.989
SweepSolver	$d \cdot g^{4/5} + p^{1/3} + g^{4/5}$	1
MPI_Testany	$p^{1/3} \cdot \log_2(g) + \log_2(d) + p^{1/3}$	1
Relearn		
Connectivity update	$\log_2(p) + n \cdot \log_2^2(n)$	0.976

Using additional measurement points for modeling does not significantly increase their accuracy. The nine base points, which we found sufficient to create accurate models, cost only about 30% of all experiments. In other words our new solution does not only allow us more flexibility in measurement selection, in the case of FASTEST we also need less than a third of the budget we would have required before.

B. Kripke

Kripke [7] is an open-source 3D Sn deterministic particle transport code. It was designed as a research tool in order to explore how different data layouts, programming paradigms and architectures effect the implementation and performance of Sn transport [7]. We already analyzed Kripke in our previous work [5], hence we now focus on showing the substantial cost reduction our new solution can achieve, while retaining the model accuracy. Therefore, we focus on the same kernels as in our previous analysis, which are: `LTimes`, `LPlusTimes`, and `SweepSolver`. These three kernels encapsulate the physics simulated by Kripke, thus they are the most interesting for a performance analysis. Additionally we take a look at the kernel `MPI_Testany`, as it accounts for a large portion of the overall runtime. The measurements we use for modeling have been conducted on Vulcan, an IBM BG/Q system at Lawrence Livermore National

Laboratory with 24,576 nodes in 24 racks. Our analysis of Kripke covers three parameters: the number of processes p, the number of direction-sets d and the number of energy groups g. For each of the selected kernels we model its execution time $T(p, d, g)$ measured in seconds as a function of the configuration parameters. In total we have 750 experiments at 150 different measurement points with five repetitions each. To create this matrix of measurement points we varied the parameter values of p (8, 64, 512, 4,096, and 32,768), d (2, 4, 6, 8, 10, and 12), and g (32, 64, 96, 128, and 160). 625 of these experiments $e = 5^{(m+1)}$ are required for modeling, in order to compare both modeling approaches with each other, while the rest of the experiments with $d = 12$ is used for evaluation.

Following the parameter-value selection heuristic, we instantiate the sparse modeler and use the three cheapest lines of five measurement points to model the effects of each parameter. In addition we use all remaining points, apart from the ones with $d = 12$, to further increase the model accuracy. In return we get the following model for `LTimes`: $4.86+2.16\cdot10^{-3}\cdot d\cdot g^{5/4}\cdot\log_2(g)+2.503\cdot d$ with $\hat{R}^2 = 0.992$, which is similar to the one we found in our previous analysis. The old model for `LTimes` is: $12.68+3.67\cdot10^{-2}\cdot d^{5/4}\cdot g$ with $\hat{R}^2 = 0.99$ [5]. We then reduce the amount of measurements to the minimum requirement of the sparse modeler to investigate the effects on accuracy. As expected this negatively affects the accuracy of all models, but as few as three additional measurements are sufficient to obtain the same models as before. Using these three additional points we get the following model for `LTimes`: $0.86+3.49\cdot10^{-3}\cdot d\cdot g^{5/4}\cdot\log_2(g)+2.09\cdot d$ with $\hat{R}^2 = 0.97$. The smaller $\hat{R}^2$ indicates a slightly less accurate model, the upside being that we require less than 1% of the cost. Table I shows the results of the sparse modeler using only the cheapest three lines of five points per parameter and the three cheapest additional measurement points that are not on this cross.

To further analyze the quality of our models we also look at their scaling. Therefore, we specifically use a measurement configuration that was not used for modeling, namely $p = 32,768$, $d = 12$ and $g = 160$. For `LTimes` we measured an average actual runtime of 143.82 seconds for this configuration. However, our performance model predicts a runtime of 199.99 seconds. This corresponds to an error of 39.8% as opposed to only 0.39% using the old modeler. An error of this magnitude could be evaluated as quite significant, though one should not forget that this configuration is several steps away from the ones used for modeling. Furthermore, our previous approach required 625 measurement points in order to create performance models of Kripke. In contrast the sparse performance-modeling only needs 80 out of these 625 points, which corresponds to a cost reduction of 99%.

C. Relearn

Relearn simulates the rewiring of connections between neurons in the brain based on the Model of Structural Plasticity (MSP) by Butz and van Ooyen [18] and employs a scalable approximation algorithm [8] to reduce the computational complexity of MSP from $\mathcal{O}(\frac{n}{p}^2)$ to $\mathcal{O}(\frac{n}{p}\log_2^2 n+p)$. Relearn has three configuration parameters, the number of processes p, the problem size n and θ, which determines the degree of approximation used by the underlying Barnes-Hut algorithm [19]. In general the developer expects an upper runtime limit of $\mathcal{O}(\theta^{-3}\frac{n}{p}\log_2^2 n+p)$ when covering all parameters [8]. However, the runtime behavior of Relearn changes depending on the value of θ. For $\theta \to 0$ the expected runtime complexity is $\mathcal{O}(\frac{n}{p}^2)$, though for larger values of θ this expectation changes to $\mathcal{O}(\frac{n}{p}\log_2^2 n + p)$. One base assumption of our modeling approach is that all configuration parameters are independent of each other. Since for Relearn this is not the case, we start our analysis considering only two parameters p and n.

We measured its performance for 25 different configurations (without repetitions) on the Lichtenberg computing cluster at the Technical University of Darmstadt. In order to obtain this amount of distinct configurations, we varied the number of processes (32, 64, 128, 256, and 512), the problem size (5,000, 6,000, 7,000, 8,000, and 9,000), and set the value of $\theta = 0.1$. Relearn is a rather small application and therefore has only a few kernels we can analyze. We have a particular interest in the `connectivity update`, as it asymptotically dominates the complexity of the computation. When using all available measurement points for modeling our old and the sparse modeler produce the exact same result, which is shown in Table I. Even when reducing the number of additional measurement points used by the sparse modeler, the accuracy does not decrease. Therefore, we can successfully reduce the modeling cost of Relearn by 85% without impacting accuracy. From the literature [8] we expect an approximate runtime behavior of $\mathcal{O}(\frac{n}{p}\log_2^2 n+p)$ for a fixed θ value. The model we find is very similar, though we are not able to predict $\frac{n}{p}$ and instead of a linear effect of p we predicted a logarithmic one. However, these are only minor inaccuracies. When analyzing the scaling of the model for $p = 512$ and $n = 9000$ the percentage error of our prediction in comparison to the actual measured value is only 11.7%.

In addition, we tried to model the performance of Relearn considering θ as a third parameter, choosing its value from (0.1, 0.15, 0.2, 0.25 and 0.3). As before, we ran the sparse modeler using only the cheapest points for each parameter, including one additional point and got the following model: $-57+7.85*\log_2 p+3.13*10^{-4}*n*\log_2 n+0.87*\theta^{-\frac{8}{3}}$ with an $\hat{R}^2 = 1$. Similar to the two parameter analysis our model is close to the expectation of the developer, this time using only about 0.58% of the budget required by the old modeling approach. However, even though the $\hat{R}^2 = 1$ indicates a good fit, the scaling analysis shows the opposite. To analyze how our model reflects the actual scaling behavior of Relearn, we measured the runtime of the `connectivity update` kernel for the configuration $p = 512$, $n = 9000$ and $\theta = 0.4$, which we did not use for modeling. For this configuration we measured an average runtime of 38.79 seconds. Our model

predicts a runtime of 60.65 seconds, which corresponds to an error rate of 56.3%. After taking a look at the measurements, we saw that the runtime of Relearn grows as the value of θ is increased. Furthermore, the runtime variation in response to scaling θ from 0.1 to 0.4 can cover several orders of magnitudes. We repeated the analysis using a different model that we obtained based on measurements with larger values of θ, that is using more expensive points and about 9.42% of the total budget. With the resulting model we are able to achieve a prediction of 30.41 seconds, which corresponds to an error of only 21.6%.

Overall, this example shows one clear limit of our approach. When there are qualitatively different behaviors depending on one parameter which affect how the other parameters impact performance, our approach is not applicable. Nevertheless, difficulties in finding a model that reflects the scaling behavior very well may actually be indicative of such a dependence, an insight that may turn out to be helpful even if the models themselves cannot be used.

VI. Related Work

Human readable performance models such as those resulting from analytical reasoning have long been acknowledged as one of the most powerful and insightful ways of describing and understanding the performance behavior of applications. Models of widely used applications [20], [21], gaining insights into complex behaviors [22]–[24], have been achieved by manually analyzing application code. Petrini et al. for example discovered through analytical modeling a large difference between actual and predicted performance caused by system noise [1]. Hoefler et al. defined a simple six-step process to create application performance models [25]. The described method leads to insight into application scaling behavior but is tedious to apply to real codes.

The main obstacle of analytical approaches is the expertise required to gain results for realistic applications and the large amount of effort this entails. Several approaches have attempted to automate the performance modeling process to make it feasible to use in practice: PALM [26], a tool that generates models after requiring users to annotate the source code with performance expressions; ASPEN [27], a language specifically developed to annotate source code with performance expressions; Vuduc et al [28], who generate the coefficients of performance models automatically while still requiring the user to select the hypotheses manually; Jayakumar et al [29], who compare kernels in applications with a database of kernels with known behavior to classify performance characteristics. Siegmund et al. consider all relevant configuration options of an application and model how their interaction affects the performance of the application [30]. However, they only treat applications monolithically and can therefore identify only the coarsest behaviors. Hoefler et al. generate multi-parameter performance models online [31]. The online nature of their approach limits the size of the search space and thus the diversity of models quite significantly. This in turn adversely affects model accuracy.

Other approaches combine traditional modeling techniques with methods from machine learning, including active and transfer learning, neural networks or decision trees, to further improve the robustness of their predictions [32]. Duplyakin et al., for example, apply active learning to suggest follow-up experiments that can help refine their initial performance models created by Gaussian process regression [33]. We, in contrast, employ reinforcement learning to derive a parameter value selection heuristic that is efficient for all kinds of HPC applications, not tuned for one specific application. Similarly, Neumann et al. use sparse grid regression to predict the performance of various HPC applications, such as molecular dynamics, climate and weather, using high-dimensional run time data [34]. Jamshidi et al. show that, in some scenarios, transfer learning can be used to reduce the cost of new performance models by transferring knowledge about performance behavior from one system to another [35]. In contrast, some approaches focus on generating models for a very specific purpose, such as learning and predicting application performance based on its input parameters using artificial neural networks [36], [37]. Kundu et al. use such an approach to model the performance of VM-hosted applications as a function of resource allocation and contention [38]. Li et al. on the other hand use a modeling approach based on regression trees to accurately predict the performance of single- and multi-core processors for unsampled points in their design space [39].

VII. Conclusion

Sparse performance-modeling in combination with an efficient parameter-value selection strategy can successfully reduce the cost of the modeling process while retaining high model accuracy. This renders taking measurements for all combinations of the selected parameter values unnecessary. Specifically, one can reduce the number of measurements required for creating an empirical performance model from exponential to polynomial, thus making the parameter value selection more flexible. We exploited this flexibility to design a generally valid heuristic for the selection of parameter values that we derived once from the knowledge of an intelligent agent. Trained on synthetic performance functions, our agent learned to select parameter-value combinations that minimize the cost and maximize the accuracy of empirical performance models. Our solution reduces the average cost of the modeling process by about 85% while retaining 92% of the model accuracy. In addition, the new flexibility in selecting parameter values at the very least simplifies and sometimes even enables the generation of empirical performance models for applications such as FASTEST, which cannot produce useful measurements or do not run at all under all combinations of practically available parameter values.

References

[1] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proc. of the ACM/IEEE Conference*

on Supercomputing, ser. (SC '03). ACM, 2003, p. 55. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050204

[2] 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.

[3] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf, "Exascaling your library: Will your implementation meet your expectations?" in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 165–175.

[4] "Extra-P – automated performance-modeling tool," www.scalasca.org/software/extra-p.

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

[6] M. Kornhaas, M. Schäfer, and D. Sternel, "Efficient numerical simulation of aeroacoustics for low mach number flows interacting with structures," *Comput Mech*, vol. 55, pp. 1143–1154, 2015.

[7] A. J. Kunen, T. S. Bailey, and P. N. Brown, "Kripke-a massively parallel transport mini-app," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2015.

[8] S. Rinke, M. Butz-Ostendorf, M.-A. Hermanns, M. Naveau, and F. Wolf, "A scalable algorithm for simulating the structural plasticity of the brain," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 251–266, 2018.

[9] P. Reisert, A. Calotoiu, S. Shudler, and F. Wolf, "Following the blind seer–creating better performance models using less information," in *European Conference on Parallel Processing*. Springer, 2017, pp. 106–118.

[10] A. Calotoiu, A. Graf, T. Hoefler, D. Lorenz, S. Rinke, and F. Wolf, "Lightweight requirements engineering for exascale co-design," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 201–211.

[11] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[12] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[14] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[15] H. V. Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems*, 2010, pp. 2613–2621.

[16] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[17] D. an Mey, S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. Malony, W. E. Nagel *et al.*, "Score-P: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010*. Springer, 2011, pp. 85–97.

[18] M. Butz and A. van Ooyen, "A simple rule for dendritic spine and axonal bouton formation can account for cortical reorganization after focal retinal lesions," *PLoS computational biology*, vol. 9, no. 10, p. e1003259, 2013.

[19] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, 1986.

[20] 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 *Proc. of the ACM/IEEE Conference on Supercomputing (SC'01)*. ACM, 2001, p. 37.

[21] M. M. Mathis, N. M. Amato, and M. L. Adams, "A general performance model for parallel sweeps on orthogonal grids for particle transport calculations," College Station, TX, USA, Tech. Rep., 2000.

[22] S. Pllana, I. Brandic, and S. Benkner, "Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art," in *Proc. of the 1st Intl. Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2007, pp. 279–284.

[23] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 336–346, February 2006. [Online]. Available: http://dx.doi.org/10.1016/j.future.2004.11.019

[24] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," *SIGMETRICS Performance Eval. Review*, vol. 32, no. 1, pp. 2–13, June 2004. [Online]. Available: http://doi.acm.org/10.1145/1012888.1005691

[25] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*, ser. SC '11. ACM, 2011, pp. 6:1–6:12. [Online]. Available: http://doi.acm.org/10.1145/2063348.2063356

[26] N. R. Tallent and A. Hoisie, "Palm: Easing the burden of analytical performance modeling," in *Proc. of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 221–230. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597683

[27] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 84:1–84:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389110

[28] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical models for empirical search-based performance tuning," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 65–94, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004041293

[29] A. Jayakumar, P. Murali, and S. Vadhiyar, "Matching application signatures for performance predictions using a single execution," in *Proc. of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015)*, May 2015, pp. 1161–1170.

[30] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 284–294. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786845

[31] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler, "Using compiler techniques to improve automatic performance modeling," in *Proc. of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT'15)*, San Francisco, CA, USA, 2015, pp. 1–12.

[32] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Proceedings of the 6th ACM/SPEC international conference on performance engineering*. ACM, 2015, pp. 145–156.

[33] D. Duplyakin, J. Brown, and R. Ricci, "Active learning in performance analysis," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 182–191.

[34] P. Neumann, "Sparse grid regression for performance prediction using high-dimensional run time data," in *Euro-Par 2019: Parallel Processing Workshops, Workshop on Performance Monitoring and Analysis of Cluster Systems, Gttingen, Germany*, ser. Lecture Notes in Computer Science. Springer, (to appear).

[35] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: An exploratory analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 497–508.

[36] 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, pp. 196–205.

[37] 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.

[38] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *ACM Sigplan Notices*, vol. 47, no. 7. ACM, 2012, pp. 3–14.

[39] B. Li, L. Peng, and B. Ramadass, "Accurate and efficient processor performance prediction via regression tree based modeling," *Journal of Systems Architecture*, vol. 55, no. 10-12, pp. 457–467, 2009.