# Following the Blind Seer — Creating Better Performance Models Using Less Information

Patrick Reisert, Alexandru Calotoiu, Sergei Shudler, and Felix Wolf

Technische Universität Darmstadt, 64289 Darmstadt, Germany
kpreisert@gmail.com, {calotoiu,shudler,wolf}@cs.tu-darmstadt.de

**Abstract.** Offering insights into the behavior of applications at higher scale, performance models are useful for finding performance bugs and tuning the system. Extra-P, a tool for automated performance modeling, uses statistical methods to automatically generate, from a small number of performance measurements, models that can be used to predict performance where no measurements are available. However, the current version requires the manual pre-configuration of a search space, which might turn out to be unsuitable for the problem at hand. Furthermore, noise in the data often leads to models that indicate a worse behavior than there actually is. In this paper, we propose a new model-generation algorithm that solves both of the above problems: The search space is built and automatically refined on demand, and a scale-independent error metric tells both when to stop the refinement process and whether a model reflects faithfully enough the behavior the data exhibits. This makes Extra-P easier to use, while also allowing it to produce more accurate results. Using data from previous case studies, we show that the mean relative prediction error decreases from 46% to 13%.

**Keywords:** parallel computing, performance tools, performance modeling

## 1 Introduction

As the computing world moves towards more and more parallelism and high-performance computing (HPC) systems become ever larger, the complexity of performance analysis is compounded. Understanding the performance of parallel programs at larger scale and getting correct insights requires prohibitive resources. Developers and users must benchmark their applications at the full extent of available parallelism to obtain the insights they desire. It requires both expensive computing time and manpower. Performance modeling offers a way to alleviate this problem by providing users with models (i.e., analytical expressions) of the application behavior. With these models users are able to predict application behavior at higher scale. One example of a performance model, which can also help uncover scalability bottlenecks, is the expression of execution time as a function of the number of processors.

We distinguish between analytical and empirical performance modeling. Analytical performance models are constructed by experts that infer the laws that

govern application behavior as a function of a pre-selected parameter (e.g., the number of processors). Not only is this a laborious process that requires intuition and small-scale tests, but it might also require experts to apply this process to every individual module of the application. Empirical performance modeling, on the other hand, infers models automatically from a relatively small number of measurements. It offers a practical way for common users to find scalability bugs and bottlenecks [5], predict performance [16,11], compare algorithmic alternatives [20], and understand the effects of resource contention [17].

*Extra-P* [1] is a tool to create such empirical performance models, primarily scaling models, with one or more model parameters [5,4]. However, in its current version it relies on a manually defined search space, requiring users either to provide a large enough space to accommodate a wider range of models or to have an initial guess as to how a model would look like. Both options have their drawbacks since the former means increased computation costs, and the latter means increased expertise on the user's part. In addition, *false positives* (i.e., models that indicate a worse behavior than there really is) can sometimes occur due to artifacts in performance measurements. Although such artifacts can usually be identified manually, actually doing it would substantially prolong the performance modeling process. In this work, we make the following contributions to address the aforementioned shortcomings:

- Automatic search-space configuration — in our new iterative model-generation approach, we configure the search space on demand and iteratively raise the accuracy of the model until no meaningful improvement can be made. In this way, we increase both the tool's ease of use and its range of application without sacrificing accuracy.
- Significant reduction of false positives — by using a heuristic to increase Extra-P's resilience to noise in the measurements, we are able to save users from wasting valuable time trying to analyze problems that do not really exist.

The remainder of the paper is organized as follows. In Section 2, we provide the background on automatic performance modeling and the current technique for model generation. Section 3 continues with a detailed description of the new, iterative refinement approach, followed by an evaluation in Section 4. Finally, we review related work in Section 5, before drawing our conclusion in Section 6.

## 2 Empirical Performance Modeling with Extra-P

In this section, we briefly introduce Extra-P and the way it generates empirical performance models.

### 2.1 The Performance Model Normal Form

A key concept underlying Extra-P is the *performance model normal form* (PMNF). The PMNF models the effect of a single parameter (predictor) $x$ on a response

variable of interest $f(x)$, typically a performance metric such as execution time or a performance counter. It is specified as follows:

$$f(x) = c_0 + \sum_{k=1}^{n} c_k \cdot x^{i_k} \cdot \log_2^{j_k}(x)$$

The PMNF allows building a function search space, which we then traverse to find the member function that comes closest to representing the set of measurements. This assumes that the true function is contained in this search space. A possible assignment of all $i_k$ and $j_k$ in a PMNF expression is called a *model hypothesis*. The sets $I, J \subset \mathbb{Q}$ from which the exponents $i_k$ and $j_k$ are chosen and the number of terms $n$ define the discrete model search space. Our experience suggests that neither $I$ and $J$ nor $n$ have to be particularly large to achieve a good fit, but although a common default set of about 40 terms can be sufficient, for some applications the search space needs to be tuned manually with the help of domain experts and application developers. Having chosen the sets, we then automatically determine the coefficients of all hypotheses using regression and choose the hypothesis with the smallest error such that we get the most likely model function.

For the above process to yield good results, the true function that is being modeled should not be qualitatively different from what the normal form can express. Discontinuities, piece-wise defined functions, and other behaviors that cannot be modeled by the normal form will lead to sub-optimal results. There are, however, many practical scenarios where programs change their behavior. For example, modern MPI implementations switch from one algorithm to another, depending on the message size, the number of processes, or the network topology. Whether an application fits within the cache or not also affects performance in a discontinuous manner. Ilyas et al. [10] introduce a novel method in Extra-P to detect such segmentation before generating empirical models. Specifically, the authors developed heuristics to successfully find segmentation in data with as few as six points, and a method to estimate the change point. In this way we can continue to use the PMNF on the level of individual segments.

### 2.2 Model Generation

Extra-P requires a set of performance profiles as input, representing runs where one or more parameters are varied. These profiles can be obtained using existing performance measurement tools. Here, we use the performance measurement system Score-P [14], which collects several performance metrics, including execution time and various hardware and software counters, broken down by call path and process. Other data sources from other performance measurement tools are equally possible and simply require some form of input format conversion.

Based on the profiles, we compute one model for each combination of target metric and call path, enabling a very fine-grained scalability analysis even of very complex applications.

Past experience has shown that as few as five different measurements for one parameter are enough for successful model generation, allowing the automatic discovery of scalability bottlenecks at very low cost.

## 3 Approach

We now introduce our novel modeling algorithm, starting with some key ideas and then presenting the algorithm as a whole. We focus here on the case of single-parameter modeling. Calotoiu et al. [4] have shown that finding multi-parameter models can be reduced to combining the best single-parameter models in different ways and selecting the combination which fits the measurements best. Therefore our approach can be used as a drop-in replacement in the multi-parameter scenario while maintaining all the benefits shown in this paper.

### 3.1 The SMAPE Metric

A key component of our new approach is the *symmetric mean absolute percentage error* (SMAPE) metric [12]. Previously, Extra-P used the *residual sum of squares* (RSS) to compare the quality of generated models and model hypotheses. Given $N$ experiments with measurements $y_i$ $(1 \leq i \leq N)$ for the parameter values $x_i$, the RSS of a model $f(x_i)$ is calculated as $\sum_{i=1}^{N}(y_i - f(x_i))^2$. One disadvantage of the RSS is that its value depends on the scale of the data that are being modeled. Smaller input values will lead to a smaller RSS, and the squaring of the residuals amplifies this problem. Furthermore, the RSS does not have a well defined range, so its value cannot be interpreted easily.

SMAPE is a scale-independent, relative error metric that overcomes the shortcomings of the RSS. It originates from time series forecasting and is defined as:

$$\frac{1}{N} \cdot \sum_{i=1}^{N} \frac{|y_i - f(x_i)|}{(|y_i| + |f(x_i)|)/2} \cdot 100\%.$$

Taken apart, it is the mean of a ratio, expressed as a percentage. The SMAPE value is always in a range of 0% to 200%, where 0% means no error at all. This makes it a helpful error metric that can be easily interpreted by a user and compared across models with different scales. In contrast to the slightly simpler MAPE metric, SMAPE does not break down when any $y_i$ is 0.

However, we still use the RSS to decide which of two model hypotheses better fits the data, simply because we use regression to fit the hypothesis to the data, which relies on the least squares method and thus optimizes the RSS metric. We have observed that in most cases[1] both metrics agree as far as the relative order is concerned, so if one hypothesis has a better RSS than another, it usually has a better SMAPE value as well.

[1] This is not generally true, which also makes the *unimodality* results presented in Section 3.2 not hold for SMAPE, even though the plots shown there usually show the same patterns when generated from SMAPE instead of the RSS.

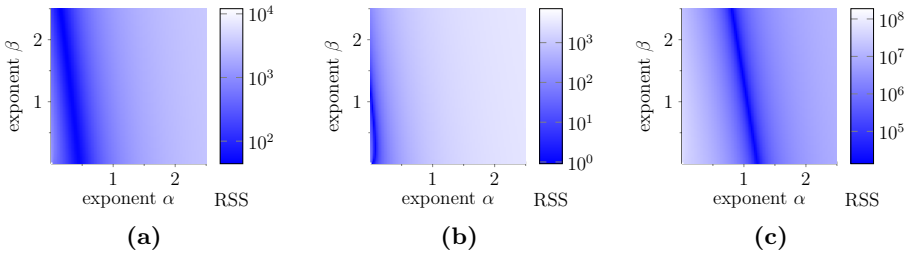


**Fig. 1:** Error of fitted simplified PMNF models for measurements from three different kernels, sampled with a resolution of $\frac{1}{40}$ for both $\alpha$ and $\beta$.

### 3.2 Revisiting the PMNF

Over time, we have accumulated experiences of common use cases and what type of analyses and configurations yield the most insightful results [5,11,16,20]. Based on these experiences, we propose to simplify the PMNF itself so that not only it fits the common use cases better, but also increases resilience to noise.

Given the cost of gathering measurements, users commonly provide less than 10 different data points per parameter. We have discovered that allowing more than one term in addition to $c_0$ almost always leads to modeling insufficiently understood behavior unless there are significantly more data points available, especially if the data is affected by noise. Therefore, we suggest to use the following *simplified PMNF*:

$$f(x) = c_0 + c_1 \cdot x^{\alpha} \cdot \log_2^{\beta}(x)$$

Since optimal values for $c_0$ and $c_1$ are determined by regression whenever $\alpha$ and $\beta$ have been fixed, the remaining challenge is to select the best exponents $\alpha$ and $\beta$, where $\alpha = 0$ and/or $\beta = 0$ is allowed. Following the observed behavior of real applications [5,11,16,20], we can restrict $\alpha < 6$ and $\beta < 3$.

To gain insights into the space spanned by $\alpha$ and $\beta$, we created heatmap plots, where each point represents the RSS of an optimal (fitted) model hypothesis with exponents $\alpha$ and $\beta$. A representative selection of such plots is presented in Figure 1. From these plots we can see that the hypotheses with minimal error run along a line, which starts on the horizontal axis ($\beta = 0$) and goes upwards and to the left, approaching the vertical axis ($\alpha = 0$). In some cases (usually for data that require a purely logarithmic model, as in Figure 1b), the line first slightly bends to the right before finally turning to the left. Thus, the function that assigns the error of the best hypothesis to a choice of $\alpha$ and $\beta$ has the following properties:

- It is unimodal (i.e., it has a single minimum, and the function value decreases as you approach that minimum from either side) over $\alpha$ for any choice of $\beta$.
- It is unimodal over $\beta$ for $\alpha = 0$.

– It is generally *not* unimodal over $\beta$ for $\alpha > 0$.

This, together with the fact that the variation along the line of minimal error is very small, is the reason for the choice of four one dimensional *slices* of this two dimensional space, along which we will search for a minimum. We define these slices as $\beta = 0, \beta = 1, \beta = 2$ and $\alpha = 0$.

Along each of these slices we can now search for appropriate values of $\alpha$ (for the slices where $\beta$ is fixed) or $\beta$ (where $\alpha$ is fixed), respectively. When we say *appropriate*, we do not necessarily mean *optimal*, as we want to find exponents that are representable by (preferably simple) fractions. We consider a fraction[2] to be simpler than another whenever it has a smaller denominator than the other.[3] We have developed an algorithm to find such exponents, which we shall introduce in the following section.

### 3.3 Iterative Refinement

Our algorithm is based on the idea that we can start with integer exponents (i.e., fractions with denominator 1) and then iteratively refine the search space by increasing the denominators while approximating the true minimum. Search space refinement has previously been proposed by Shudler et al. [16] who suggested repeated halving of an initial interval, which resulted in denominators that are always powers of two. However, a computational kernel simulating a three dimensional process, for example, can actually require an exponent of $\frac{n}{3}$ to model its complexity.

For the sake of presentation, let us now first look only at the slice $\beta = 0$, where no logarithmic term is involved and the algorithm tries to find an appropriate value for $\alpha$. We shall later expand on how the algorithm deals with multiple slices. First, all hypotheses with integer exponents $\alpha = 0, \ldots, 5$ are computed and compared. The exponent leading to the best model is stored, and its successor and predecessor are used as initial upper and lower bounds, respectively. After this initialization, the actual iterative refinement process, presented in Figure 2, starts. It constitutes a variant of the golden section search [15], but uses the *mediant* instead of the golden section to determine new candidate exponents, for reasons explained below. The mediant of two fractions $\frac{n_1}{d_1}$ and $\frac{n_2}{d_2}$ is defined as $\frac{n_1+n_2}{d_1+d_2}$ and has the property that it always lies in between the two original fractions [8]. For example, the mediant of $\frac{1}{2}$ and 1 (represented as $\frac{1}{1}$) is $\frac{2}{3}$.

In every iteration, two new hypotheses are computed from the two mediants in between the currently best hypothesis and each of the two bounds, and their errors are compared to the best hypothesis. If the left mediant has the smallest error, we cut off the right part of the search space by using the left mediant for the new best hypothesis. If the right mediant is the winner, we do the same on the other side. If none of the mediants has a smaller error, then the best hypothesis remains the same and we use the two mediants as new upper and lower bounds.

[2] We are only concerned with *fully reduced* (also called *irreducible*) fractions here.
[3] This is in line with a simplicity metric presented by Guthery [8, p. 163].

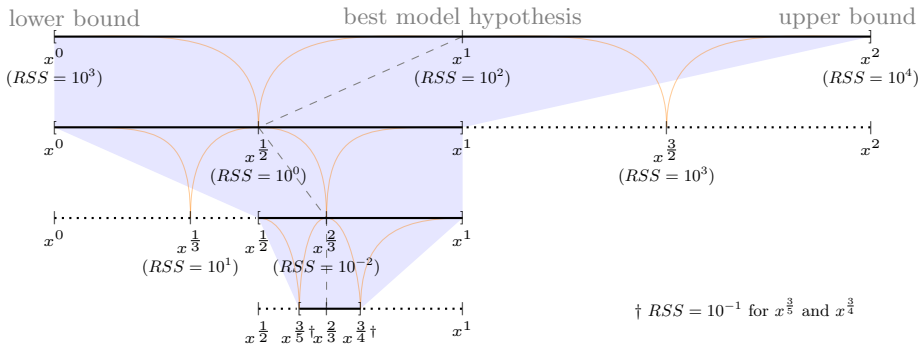


**Fig. 2:** Example showing three iterations of our refinement algorithm. In the first iteration, the search space is cut on the right side, because the left mediant has the smallest error; in the second iteration, it is cut on the left side; in the third iteration, it is cut on both sides, because no mediant has a smaller error. Orange curved lines indicate the calculation of the mediant.

If we kept computing mediants in this way, we would obtain a sequence of fractions with ever increasing denominators, ever more accurately approximating the true minimum. This sequence is a path in the *Stern–Brocot tree*, an infinite binary tree that enumerates all positive rational numbers [8]. Hence, when we arrive at a fraction $\frac{n_i}{d_i}$ in our algorithm, no fraction in between with a denominator less than $d_i$ was missed, because such a fraction would have appeared earlier along the path in the tree.

However, we want to stop after a small number of iterations (usually 1 to 3) to keep the exponents readable and more intuitive. In order to decide when to stop, we can draw on the benefits of the SMAPE metric, as it allows our algorithm to make decisions based on the relative improvement of its value. Thus, the SMAPE improvement will serve as a termination criterion in our algorithm. Moreover, we have observed that most models which have a SMAPE value that is not at least twice as good (where smaller is better) as that of the constant model do not justify the choice of a non-constant model. After manual inspection of the underlying data, in the vast majority of cases the data appears roughly constant, with small deviations in both directions that can be explained by noise. Since any model will fit the data better than the constant model in such a case (because the constant model cannot bend in any way), we penalize the choice of the non-constant model to reduce false positives. We can now outline the full algorithm:

*Step 1.* For each slice, find and remember the best integer exponent.

*Step 2.* Refine each slice according to the previously described method. All slices execute one iteration of the algorithm before the next iteration is started.

In each iteration, the best model hypothesis among all slices is considered as a candidate for the globally best hypothesis. To be accepted it needs to provide an improvement over the previously accepted best hypothesis that is large enough to justify a finer grained exponent. We use SMAPE to measure this improvement and define an *acceptance threshold* of 1.5 (i.e., an improvement of at least 50%) per iteration. The search terminates when in a single iteration no slice improves its SMAPE value by at least a factor of 2 (we call this the *termination threshold*).

*Step 3.* After the iterative refinement has terminated, the winner hypothesis from the previous step is compared to the constant model and accepted only if the SMAPE value has improved by at least a factor of 2 (the *non-constancy threshold*), otherwise, as discussed earlier, it is rejected in favor of the simpler constant model.

## 4 Evaluation

We have evaluated our algorithm in two different ways. First, to gauge the accuracy of the algorithm, we evaluated it on synthetic data with known underlying functions. We compared the results with the output of the original algorithm, for which the following default search space was used, matching the one suggested in the latest publication by Calotoiu et al. [4]: $n = 2$, $I = \{\frac{0}{4}, \frac{1}{4}, \ldots, \frac{12}{4}\}$, and $J = \{0, 1, 2\}$. Second, to understand how helpful the new algorithm is in practice and the improvements it offers, we evaluated it on measured data collected in previous case studies. The latter results, however, are more difficult to interpret because the ground truth (i.e., the true underlying functions, which the modeling ideally should recover) is with few exceptions practically inaccessible.

### 4.1 Synthetic Data

Figure 3 presents evaluation results based on randomly generated synthetic data. Because we have found most real models to be constant or very simple—the common case that our method is primarily designed for and that matches the asymptotic complexities of many known algorithms—we defined different classes of functions based on the following classification of terms:

- Common: $x$, $x^2$, $x^3$, $\log_2(x)$
- Rare: $x^{\frac{i}{2}}$ for $i \in \{1, 3, 5\}$, $x^{\frac{i}{3}}$ for $i \in \{1, 2, 4, 5, 7, 8\}$, $\log_2^2(x)$
- Exotic: $x^{\frac{i}{4}}$ for $i \in \{1, 3, \ldots, 11\}$, $x^{\frac{i}{5}}$ for $i \in \{1, \ldots, 14\} \setminus \{5, 10\}$, $\log_2^{\frac{1}{2}}(x)$, $\log_2^{\frac{3}{2}}(x)$

A function classified as *rare* may contain terms classified as *common*, but it must contain at least one *rare* term. Likewise, *exotic* functions might contain terms from the other classes, but must contain at least one of the *exotic* terms, which are terms that we have not observed so far in real applications but we assume that they could occur.

For each of the seven distinct cases shown in Figure 3, we generated 1000 random functions and evaluated them for each of four different sets of $x$ values

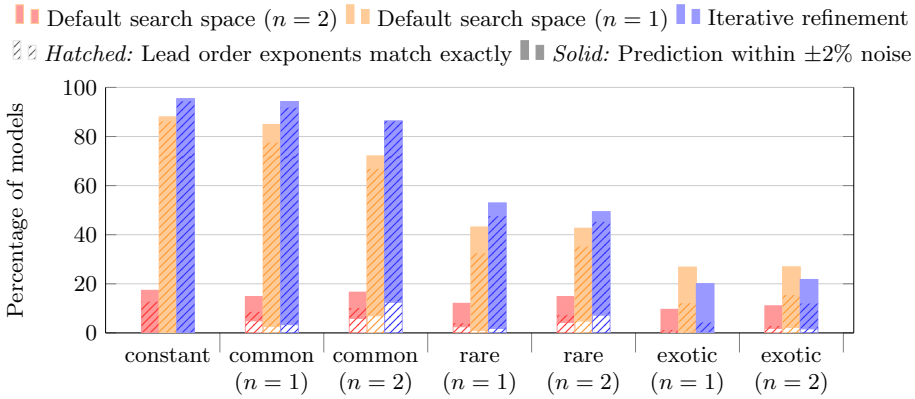


**Fig. 3:** Comparison of the original and our new algorithm using values of randomly generated functions with ±2% of noise as input. The functions are built according to the PMNF with $n = 1$ or $n = 2$, and their coefficients $c_0$, $c_1$ and $c_2$ are calculated by sampling $a \in [-2, 3]$ uniformly and then computing $10^a$.

that are representative for Extra-P's use cases ({2, 4, 8, 16, 32}, {8, 16, 32, 64, 128}, {32, 64, 128, 256, 512}, and {128, 256, 512, 1024, 2048}). To each function value we added ±2% of noise, drawn from a uniform distribution, before using the values as input for the modeling algorithms.

We then checked whether (1) the resulting model's exponents were matching exactly the expected lead order exponents of the input function and/or (2) the model's prediction for an $x$ value that is four times as large as the largest value used for modeling is within the ±2% noise level of the actual function value. For functions with two additive non-constant terms, we define the lead order term to be the one that contributes to the function more than the other when evaluated at an $x$ value that is four times as large as the largest one used for modeling.

For constant, common, and rare functions, our algorithm shows improvements upon the original one with respect to both the amount of exactly matched exponents as well as the number of accurate predictions, even when that algorithm is restricted to model only a single term. While the results for the exotic functions are less favorable, the old algorithm is not able to produce significantly better results with either $n = 1$ or $n = 2$, and we still have to find such functions in practice.

### 4.2 Case Studies

We used measurements from previous case studies to evaluate our new algorithm on measured data. The measurements include a variety of call paths (i.e., kernels) and different metrics, such as runtime, number of function calls, memory footprint, and network traffic. Whereas the evaluation using synthetic data

**Table 1:** Comparison of the original and our improved algorithm, using data from previous case studies, showing the quality of predictions of the last data point when that point is not used for modeling.

| Benchmark | Number of points | Model count | Model predictions (percentage of all models) | | | Mean relative prediction error [%] | |
|---|---|---|---|---|---|---|---|
| | | | better | same | worse | before | now |
| Sweep3D [5] | 7 | 96 | 26.04 | 56.25 | 17.71 | 17.26 | 6.31 |
| HOMME [5] | 9 | 670 | 18.81 | 68.51 | 12.69 | 3.69 | 3.03 |
| MILC [5] | 9 | 1496 | 30.95 | 56.48 | 12.57 | 36.71 | 14.53 |
| UG4 [20] | 5 | 2026 | 52.62 | 38.01 | 9.38 | 68.30 | 15.58 |
| MPI collect. [16] | 7–8 | 26 | 65.38 | 7.69 | 26.92 | 52.53 | 15.89 |
| BLAST [4] | 5 | 103 | 31.07 | 41.75 | 27.18 | 34.92 | 10.38 |
| Kripke [4] | 5 | 36 | 36.11 | 38.89 | 25.00 | 33.05 | 8.32 |
| Total | 5–9 | 4453 | 39.12 | 49.11 | 11.77 | 45.71 | 12.97 |

gives us confidence that the method works in principle, the evaluation with real applications shows its practical benefit.

The results of the comparison, which are presented in Table 1, show that when the last (i.e., largest) measured data point is excluded from the data used to calculate the model, the model produced by our new algorithm allows for a better prediction of the last point in 19%–65% of the cases, which corresponds to 53%–85% of those models that changed in each benchmark. Although some predictions do get worse, the mean relative prediction error, which we computed using the SMAPE formula (but this time averaging over the last data point of all models instead of all data points of a single model), decreases across all applications, in all but one case even significantly. We use this metric here for reasons similar to those discussed in Section 3.1: We need a scale-independent error metric because the scale of the data varies heavily among the different benchmarks and modeled performance metrics.

Not shown in the table is the number of models that are constant, which has considerably increased in every single case study (from 44% to 76% overall). Because the synthetic evaluation has shown that our new algorithm is able to recognize constant functions more reliably, this indicates that the previous algorithm might have modeled noise or tried to fit a PMNF function to inaccurate measurements.

## 5 Related Work

In recent years, performance modeling of HPC applications has become a very active field of study [3,4,6,7,13,17,19]. Previous work explored regression-based approaches for predicting program scalability [2,7]. In one case [2], Barnes et al. used linear regression to fit the measured execution time to a second-order

polynomial. In a different study [7], the authors used Active Learning to improve regression-based models they produced from measured data. Active Learning is a group of machine learning techniques in which the learner can decide to query the information source, at some additional cost, to label a datapoint that is otherwise hard to label. It means that to improve the initial performance model the technique would decide in which configuration it should run the next experiment so that the result produces the best improvement in the model with minimal cost. Active Learning complements our methodology by starting with a small set of measurements and deciding which experiments to run next. In contrast to regression-based techniques, our methodology produces simplified PMNF models that are based on combinations of logarithmic and linear terms one often finds in common algorithms. It provides the user with more insights into the behavior of the modeled applications.

A number of previous studies [3,9] relied on semi-analytical modeling to produce performance models. However, unlike semi-analytical approaches, empirical performance modeling focuses on a common user, who might not have the necessary expertise to construct the initial model. The refinement algorithm advances this concept even further by relieving the user from the burden of providing the terms for the model search space.

Aspen [18] and Palm [19] are both top-down analytical modeling approaches. The former offers a domain-specific language and the latter source-code annotations. While both of these approaches produce accurate models, they are not empirical and therefore can miss potential bottlenecks and scalability limitations.

Meswani et al. [13] present a different modeling approach that focuses only on hybrid CPU-GPU systems. Another approach which focuses only on shared memory machines is ESTIMA [6]. It measures the number of stalled cycles on a small number of cores and estimates the slowdown on a higher number of cores. Although both approaches predict future scalability of applications, they do not offer the same degree of flexibility as Extra-P does.

## 6 Conclusion

In this paper, we propose a novel algorithm for empirical performance modeling as part of the Extra-P tool. In contrast to previous work, we remove the need for a predefined search space and also significantly reduce the number of false positives by being more resilient to noisy measurements of constant behavior. Yet most of the models generated with our algorithm are able to make predictions that are equally or even more accurate than before, which we demonstrate with experiments using both synthetic and real data. Thus, our work opens the way for a performance modeling workflow that is more automated than ever and equips developers with a tool that helps them efficiently find scalability bugs in large applications.

*Acknowledgements.* This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 *Software for Exascale*

*Computing* (SPPEXA) and the Programme *Performance Engineering for Scientific Software*. Additional support was provided by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH16008, and by the US Department of Energy under Grant No. DE-SC0015524. Finally, we would like to thank the University Computing Center (Hochschulrechenzentrum) of TU Darmstadt for providing us with access to the Lichtenberg Cluster.

## References

1. Extra-P – Automated Performance-Modeling Tool, www.scalasca.org/software/extra-p
2. Barnes, B.J., Rountree, B., Lowenthal, D.K., Reeves, J., de Supinski, B., Schulz, M.: A Regression-based Approach to Scalability Prediction. In: Proc. of the International Conference on Supercomputing (ICS). pp. 368–377. ACM (2008)
3. Bauer, G., Gottlieb, S., Hoefler, T.: Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application Su3_Rmd. In: Proc. of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 652–659. IEEE Computer Society (2012)
4. Calotoiu, A., Beckingsale, D., Earl, C.W., Hoefler, T., Karlin, I., Schulz, M., Wolf, F.: Fast Multi-Parameter Performance Modeling. In: Proc. of the 2016 IEEE International Conference on Cluster Computing (CLUSTER). pp. 1–10. IEEE Computer Society (2016)
5. Calotoiu, A., Hoefler, T., Poke, M., Wolf, F.: Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In: Proc. of the 2013 ACM/IEEE Conference on Supercomputing (SC). pp. 45:1–45:12. ACM (2013)
6. Chatzopoulos, G., Dragojević, A., Guerraoui, R.: ESTIMA: Extrapolating Scalability of In-memory Applications. In: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 27:1–27:11. ACM (2016)
7. Duplyakin, D., Brown, J., Ricci, R.: Active Learning in Performance Analysis. In: Proc. of the 2016 IEEE International Conference on Cluster Computing (CLUSTER). pp. 182–191. IEEE Computer Society (2016)
8. Guthery, S.B.: A Motif of Mathematics. Docent Press (2011)
9. Hoefler, T., Gropp, W., Thakur, R., Träff, J.L.: Toward Performance Models of MPI Implementations for Understanding Application Scaling Issues. In: Proc. of the European MPI Users' Group Meeting (EuroMPI). pp. 21–30. Springer (2010)
10. Ilyas, K., Calotoiu, A., Wolf, F.: Off-road performance modeling – how to deal with segmented data. In: Proc. of the 23rd Euro-Par Conference, Santiago de Compostela, Spain. pp. 1–12. Lecture Notes in Computer Science, Springer (Aug 2017), (accepted)
11. Iwainsky, C., Shudler, S., Calotoiu, A., Strube, A., Knobloch, M., Bischof, C., Wolf, F.: How Many Threads will be too Many? On the Scalability of OpenMP Implementations. In: Proc. of the 21st Euro-Par Conference. LNCS, vol. 9233, pp. 451–463. Springer (2015)
12. Kreinovich, V., Nguyen, H.T., Ouncharoen, R.: How to Estimate Forecasting Quality: A System-Motivated Derivation of Symmetric Mean Absolute Percentage Error (SMAPE) and Other Similar Characteristics. Tech. Rep. Paper 865, University of Texas at El Paso (2014)

13. Meswani, M.R., Carrington, L., Unat, D., Snavely, A., Baden, S., Poole, S.: Modeling and Predicting Performance of High Performance Computing Applications on Hardware Accelerators. International Journal of High Performance Computing Applications 27(2), 89–108 (2013)
14. an Mey, D., Biersdorff, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S.S., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Unified Performance Measurement System for Petascale Applications. In: Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V. pp. 85–97. Gauß-Allianz, Springer (2010)
15. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes 3rd Edition: The Art of Scientific Computing. Cambridge University Press, 3 edn. (2007)
16. Shudler, S., Calotoiu, A., Hoefler, T., Strube, A., Wolf, F.: Exascaling Your Library: Will Your Implementation Meet Your Expectations? In: Proc. of the International Conference on Supercomputing (ICS). pp. 165–175. ACM (2015)
17. Shudler, S., Calotoiu, A., Hoefler, T., Wolf, F.: Isoefficiency in Practice: Configuring and Understanding the Performance of Task-based Applications. In: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 1–13. ACM (2017)
18. Spafford, K.L., Vetter, J.S.: Aspen: A Domain Specific Language for Performance Modeling. In: Proc. of the 2012 ACM/IEEE Conference on Supercomputing (SC). pp. 84:1–84:11. IEEE Computer Society Press (2012)
19. Tallent, N.R., Hoisie, A.: Palm: Easing the Burden of Analytical Performance Modeling. In: Proc. of the 28th ACM International Conference on Supercomputing (ICS). pp. 221–230. ACM (2014)
20. Vogel, A., Calotoiu, A., Strube, A., Reiter, S., Nägel, A., Wolf, F., Wittum, G.: 10,000 Performance Models per Minute - Scalability of the UG4 Simulation Framework. In: Proc. of the 21st Euro-Par Conference. LNCS, vol. 9233, pp. 519–531. Springer (2015)