# Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes

Alexandru Calotoiu
German Research School
for Simulation Sciences
RWTH Aachen University
Aachen, Germany
a.calotoiu@grs-sim.de

Torsten Hoefler
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Marius Poke, Felix Wolf
German Research School
for Simulation Sciences
RWTH Aachen University
Aachen, Germany
{m.poke,f.wolf}@grs-sim.de

## ABSTRACT

Many parallel applications suffer from latent performance limitations that may prevent them from scaling to larger machine sizes. Often, such scalability bugs manifest themselves only when an attempt to scale the code is actually being made—a point where remediation can be difficult. However, creating analytical performance models that would allow such issues to be pinpointed earlier is so laborious that application developers attempt it at most for a few selected kernels, running the risk of missing harmful bottlenecks. In this paper, we show how both coverage and speed of this scalability analysis can be substantially improved. Generating an empirical performance model automatically for each part of a parallel program, we can easily identify those parts that will reduce performance at larger core counts. Using a climate simulation as an example, we demonstrate that scalability bugs are not confined to those routines usually chosen as kernels.

## Keywords

scalability, performance analysis, performance modeling, Scalasca

## 1. INTRODUCTION

When scaling their codes to larger numbers of processors, many HPC application developers face the situation that all of a sudden a part of the program starts consuming an excessive amount of time. Unfortunately, discovering latent scalability bottlenecks through experience is painful and expensive. Removing them requires not only potentially numerous large-scale experiments to track them down, prolonged by the scalability issue at hand, but often also major code surgery in the aftermath. All too often, this happens at a moment when the manpower is needed elsewhere. This is especially true for applications on the path to exascale, which have to address numerous technical challenges simultaneously, ranging from heterogeneous computing to resilience. Since such problems usually emerge at a later stage of the development process, dependencies between their source and the rest of the code that have grown over time can make remediation even harder. One way of finding scalability bottlenecks earlier is through analytical performance modeling. An analytical scalability model expresses the execution time or other resources needed to complete the program as a function of the number of processors. Unfortunately, the laws according to which the resources needed by the code change as the number of processors increases are often laborious to infer and may also vary significantly across individual parts of complex modular programs. This is why analytical performance modeling—in spite of its potential—is rarely used to predict the scaling behavior before problems manifest themselves. As a consequence, this technique is still confined to a small community of experts.

If today developers decide to model the scalability of their code, and many shy away from the effort, they first apply both intuition and tests at smaller scales to identify so-called *kernels*, which are those parts of the program that are expected to dominate its performance at larger scales. This step is essential because modeling a full application with hundreds of modules manually is not feasible. Then they apply reasoning in a time-consuming process to create analytical models that describe the scaling behavior of their kernels more precisely. In a way, they have to solve a chicken-and-egg problem: to find the right kernels, they require a pre-existing notion of which parts of the program will dominate its behavior at scale—basically a model of their performance. However, they do not have enough time to develop models for more than a few pre-selected candidate kernels, inevitably exposing themselves to the danger of overlooking unscalable code.

In this paper, we introduce a novel tool that eliminates this dilemma. Instead of modeling only a small subset of the program manually, we generate an empirical performance model for each part of the target program automatically, significantly increasing not only the coverage of the scalability check but also its speed. All it takes to search for scalability issues even in full-blown codes is to run a manageable number of small-scale performance experiments, launch our tool, and compare the extrapolated performance of the worst instances to expectations. To make this possible, we exploit


SC '13, November 17 - 21 2013, , USA

http://dx.doi.org/10.1145/2503210.2503277

several assumptions:

1. We take advantage of the observation that the space of the function classes underlying these models is usually small enough to be searched by a computer program. An iterative refinement process maximizes both the efficiency of the search and the accuracy of our models.
2. We abandon model accuracy as the primary success metric and rather focus on the binary notion of *scalability bugs*. Similar to a thread checker, every scalability problem we identify is a success as long as false positives that send us in a wrong direction are rare. False negatives are, of course, undesirable but acceptable as long as the number of scalability bugs we find justifies the effort.
3. We create requirements models alongside execution-time models. A comparison between the two can illuminate the nature of a scalability problem. Also, the generation of requirements models is less affected by performance variations.

Given that our tool relies on standard performance-measurement infrastructure, the extra software that we developed is so lightweight that it is economically feasible to provide it in production-level quality. Finally, we generate not only a list of potential bugs but human-readable models that can be further elaborated to conduct a variety of deeper analyses such as investigating the possibility of cache spills.

The remainder of the paper is structured as follows. In the next section, we outline our approach and present the details of the underlying mathematical framework. We then briefly explain its integration in a production-level performance analysis toolset in Section 3. In Section 4, we show in experiments that we cannot only reproduce models that exist in the literature, but also find scalability problems in code for which no model is available. Finally, we compare our approach to earlier work in Section 5, before we summarize our findings and discuss further steps in Section 6.

# 2. APPROACH

The primary objective of our approach is the identification of *scalability bugs*. A scalability bug is a part of the program whose scaling behavior is unintentionally poor, that is, much worse than expected. As computing hardware moves towards exascale, developers need early feedback on the scalability of their software design so that they can adapt it to the requirements of larger problem and machine sizes. Although in general our method can also be applied to strong scaling, this paper concentrates on *weak scaling*. In addition to searching for performance bugs, the models our tool produces also support projections that can be helpful when applying for the compute time needed to solve the next larger class of problems. Finally, because we model both execution time and requirements alongside each other, our results can also assist in software-hardware co-design or help uncover growing wait states. Note that although our approach can be easily generalized to cover many programming models, this paper focuses exclusively on message-passing programs.

The input of our tool is a set of performance measurements on different processor counts $\{p_1, \ldots, p_{max}\}$ in the form of parallel profiles. The output of our tool is a list of program regions, ranked by their predicted execution time at a target scale of $p_t > p_{max}$ processors. We call these regions *kernels* because they define the code granularity at which we generate our models. Users who want to know their application's scalability at exascale will likely choose $p_t \gg p_{max}$. In our evaluation in Section 4, we demonstrate reasonably accurate projections for $p_t = 128 \cdot p_{max}$. If only the asymptotic behavior is of interest (i.e., $p_t \to \infty$), the ranking can be based exclusively on the growth function class itself. We do not claim that our ranking will be 100% accurate—especially when the ranking is based on the times predicted for a specific scale $p_t \gg p_{max}$. However, it will usually be good enough to draw attention to the right kernels. Of course, false negatives, which are program regions that are not identified because they wrongly appear too far at the bottom, may occur if a phenomenon relevant at scale is not captured in our data. Nevertheless, given that we provide confidence information along with our models, we assert that false positives are extremely unlikely. In a final step, the user needs to compare the projected with the expected behavior for each kernel. This has to be done manually because we cannot predict user expectations nor can we assume that the user has precise expectations for every kernel we identify. To formulate expectations users may, for example, rely on the isoefficiency metric [13].

In general, our underlying mathematical framework can accommodate more or simply different independent parameters than just the number of processors $p$. For example, in Section 4.2 we show how to obtain highly accurate performance predictions when varying the problem size per process while keeping $p$ constant. Nevertheless, the emphasis of this study is on varying $p$ only, while assuming that all other input parameters either depend on $p$ or remain stable. Note that violations of this assumption do not preclude the application of our method, they simply lower the accuracy with which we identify scalability bugs.

Figure 1 gives an overview of the different steps necessary to find scalability bugs, whose details we explain further below. To ensure a statistically relevant set of performance data, profile measurements may have to be repeated several times—at least on systems subject to jitter. This is done in the optional statistical quality control step. Once this is accomplished, we apply regression to obtain a coarse performance model for every possible program region. These models then undergo an iterative refinement process until the model quality has reached a saturation point. To arrange the program regions in a ranked list, we extrapolate the performance either to a specific target scale $p_t$ or to infinity, which means we use the asymptotic behavior as the basis of our comparison. Finally, if the granularity of our program regions is not sufficient to arrive at an actionable recommendation, performance measurements, and thus the kernels under investigation, can be further refined via more detailed instrumentation.

## 2.1 Performance measurements

We generate the parallel profiles needed as input to our tool using Scalasca [12], which records the execution time plus various performance counters, including hardware counters such as the number of floating point instructions and software counters such as the number of bytes an MPI function sends or receives. All metrics are broken down by call path

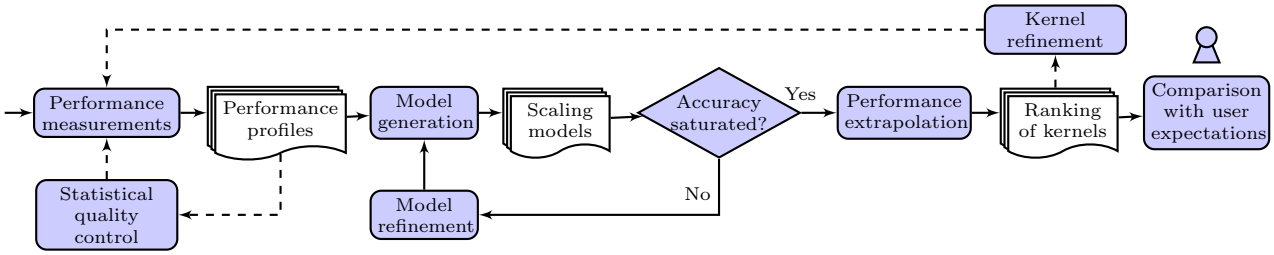


Figure 1: Workflow of scalability-bug detection. Solid boxes represent actions or transformations, and banners their inputs and outputs. Dashed arrows indicate optional paths taken after user decisions.

and process. We define a call path as a program region plus its calling context such as $main \to foo \to MPI\_Send$. Going beyond purely static program regions will allow scalability problems to be pinpointed more precisely. To keep the code covered by a call path small, performance metrics are collected with exclusive semantics, that is, for each call path without including its children. Scalasca's default instrumentation delivers performance data at the granularity of functions as call-chain elements. However, manual instrumentation can be added to distinguish lower-level constructs such as loops, which may be needed during kernel refinement. In any case, at the default granularity, computational call paths are already clearly distinguished from communication, ensuring that communication is modeled separately from computation. In the next step, we collapse the process dimension via maximum reduction, keeping one value per call path and metric. The target function we want to model is thus the maximum wall-clock time per call path. We choose the maximum because it enables us to capture bottlenecks even if they are confined to a small subset of the processes. However, the user can also select other reduction functions such as the arithmetic or geometric mean or median that are less sensitive to statistical outliers but may pose the risk of hiding performance bugs. Of course, aggregation across all processes assumes bulk synchronous parallel (BSP) programs. For irregular programs, such as some graph computations or task-based execution models, we would have to resort to a hierarchical scheme that only summarizes subsets of processes with similar behavior. Of course, this would increase the number of performance models.

The metrics we collect include both requirements-based metrics and time-based metrics. Requirements-based metrics such as the number of arithmetic operations or the number of messages sent or received are usually a function of the execution configuration and therefore more or less deterministic. As a welcome side effect, they are largely immune to system noise [17]. We call them *requirements-based* because they reflect the requirements of the program rather than the resources mustered to satisfy them. Requirements-based metrics play an important role in our approach because—supported by their deterministic nature—they can often be used to determine the function class underlying our performance models more easily. Frequently, this function class is known a priori or a known function can be used as a good approximation. Time-based metrics, such as the wall-clock time spent in communication, in contrast, are needed to determine the coefficients of our model functions or the model functions themselves when they cannot be expressed as a function of requirements-based metrics. We discuss discrepancies between time-based and requirements-based models later in Section 2.3. In any case, time-based metrics are indispensable when we want to extrapolate execution times to a specific $p_t$.

## 2.2 Statistical quality control

On many systems, performance measurements are subject to serious run-to-run variation as a consequence of OS jitter, network contention, and other nondeterministic factors. Although not an intrinsic element of our approach, we anticipate such noise and account for it by calculating confidence intervals. For this purpose, the user can repeat measurements until the variance stabilizes. Then, our tool performs a final check to ensure that the deviation is not prohibitive. An extreme example of such a case would be a system where the deviation across repeated measurements with the same input configuration is greater than the difference across different configurations, rendering any subsequent modeling meaningless.

## 2.3 Model generation

Model generation forms the core of our method. Below, we explain the different aspects and their underlying ideas in more detail.

*Arriving at a model hypothesis.* When generating performance models, we exploit the observation that they are usually composed of a finite number $n$ of predefined terms, involving powers and logarithms of $p$ (or some other parameter):

$$f(p) = \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot log_2^{j_k}(p) \qquad (1)$$

This representation is, of course, not exhaustive, but works in most practical scenarios since it is a consequence of how most computer algorithms are designed. We call it the *performance model normal form* (PMNF). Moreover, our experience suggests that neither the sets $I, J \subset \mathbb{Q}$ from which the exponents $i_k$ and $j_k$ are chosen nor the number of terms $n$ have to be arbitrarily large or random to achieve a good fit. Thus, instead of deriving the models through reasoning, we only need to make reasonable choices for $n$, $I$, and $J$ and then simply try all assignment options one by one. A possible assignment of all $i_k$ and $j_k$ in a PMNF expression is called a *model hypothesis*. Trying all hypotheses one by one means that for each of them we find coefficients $c_k$ with optimal fit. Then we apply cross-validation [29] to select

the hypothesis with the best fit across all candidates. Of course, the computational effort required to calculate our model depends on $n$, $|I|$, and $|J|$. On the other hand, a larger number $n$ of constituent terms does not necessarily imply a better model. To strike a good balance, our models are generated in an iterative refinement process, which we outline in Section 2.4. As a default, we select $n = 5$, $I = \left\{\frac{0}{2}, \frac{1}{2}, \frac{2}{2}, \frac{3}{2}, \frac{4}{2}, \frac{5}{2}, \frac{6}{2}\right\}$, and $J = \{0, 1, 2\}$. Given that the tuples $(i, j) \in I \times J$ can be ordered by their corresponding asymptotic behavior, our choices for $I$ and $J$ reflect a range of behaviors from perfect to poor scalability in 21 steps. Scalability worse than $p^3 \cdot log^2(p)$ is not distinguished. If the behavior of the application is already known to some degree, the sets $I$ and $J$ can be extended to provide more detail in a given range. For example, adding more fractional exponents in the (0,1) interval such as $\left\{\frac{1}{4}, \frac{1}{3}, \frac{2}{3}, \frac{3}{4}\right\}$ for applications where the goal is not to find out whether they scale at all but rather how well they scale can provide additional insight.

***Modeling requirements alongside time.*** Another key aspect of our approach is that we build requirements models alongside execution-time models and compare them to each other. In essence, we build empirical requirements models, which we subsequently try to match with the measured execution time. As we will explain further below, the quality of this match can reveal important facts about the application—regardless of whether the models are in agreement or show discrepancies. Since requirements-based metrics are much less prone to jitter than time-based metrics, they are much more likely to capture the asymptotic behavior correctly. This is because requirements models are closer to algorithmic complexities than empirical models derived exclusively from time-based metrics. In fact, an empirical requirements model alone can provide valuable insights when compared to developer expectations. The general idea of a requirements model is to account for all major cost factors. For computational call paths, these are the different types of operations such as floating-point operations, loads, stores, etc. Of course, in empirical models these operations have to be mapped onto the instruction set and the hardware counters available on the target system. For communication call paths, the cost factors are the number and the size of messages. We measure them using the standardized PMPI interface, which is portable across all MPI implementations.

We try to choose our metrics such that each cost factor is counted only once, although a certain degree of overlap can be tolerated. For the sake of simplicity, we assume that the total costs within a certain cost category (e.g., floating-point operations) rise linearly with the corresponding metric. That is, twice as many operations will take twice as long. Of course, some costs may disappear through latency hiding, prefetching etc. Nevertheless, we believe that this inaccuracy matters much less when our primary question only refers to the behavior at scale. Taken to an extreme, the asymptotic complexity of the scaling function does not improve simply because we can execute four floating-point operations and two loads or stores at once.

To express the execution time of a call path as a function of its requirements, we distinguish between local and global operations. Recall that the maximum aggregation across all processes we perform essentially results in a process-local metric. For the execution time of local operations, which cover computation and point-to-point communication, we therefore assume a linear relationship. This is because all processes can carry out their local operations in parallel. For example, the time it takes to send a certain number of messages $m$ of size $s$ is $t = m(c_1 + c_2 \cdot s)$. Of course, the number of messages a process sends may depend on the input configuration. For global operations, which currently cover only collectives, we resort to typical known algorithmic complexities. For example, to model the global broadcast of a message of size $s$, we would try $t = c_1 + c_2 \cdot s + c_3 \cdot \log(p)$. Lacking records of individual message sizes in our profiles, we rely on averages for $s$.

Thus, for a given call path, we first model each of the requirements-based metrics separately, generating a full regression model for each metric. To match those with the execution time, we add the resulting models to obtain a model hypothesis for this particular call path. Finally, we map this hypothesis onto the execution time, determining new time-aligned coefficients through regression. Now, we calculate the deviation between the two. If they are in good agreement, the user can draw conclusions about the primary factor contributing to the time (e.g., number of messages or floating-point operations). If not, the user can regard this as a sign that the execution time does not exclusively depend on the requirements and may be prolonged by wait states. An extreme example is serialized code. There, the maximum execution time across all processes, which forms the basis of our analysis, may remain constant, while waiting time dilates the overall execution linearly with $p$. Another example is collectives whose execution times are extended by jitter [17]. A very common source of wait states is load imbalance, a problem that particularly affects irregular codes.

***Determining the fit.*** To measure the fit we use cross-validation. This involves dividing the performance data into training and evaluation sets (i.e., sets of profiles). We use the training sets to create the model and the evaluation sets to calculate the fit. This has the advantage of protecting against overfitting, which may result in a model that tightly fits the input data points but does not accurately represent the asymptotic behavior. This problem is often encountered when fitting a polynomial of an order higher than or equal to that of the number of available data points. Specifically, we apply $k$-fold cross-validation, including the variants of holdout and leave-one-out [4, 14, 29, 41]. $k$-fold cross-validation divides the input into $k$ sets of equal size. One of the sets is used for validation and all others for training. The holdout method divides the data into two sets of equal size ($k = 2$), using one as the training and the other as the evaluation set. The leave-one-out-cross-validation (LOOCV) method uses, as its name suggests, a single data point (i.e., profile) for validation and all others for training. LOOCV is the slowest because it requires as many cross-validation passes as there are data points. On the other hand, it provides good results for very small numbers of data points ($< 10$). Holdout is faster but requires more data points. We implemented the general algorithm allowing k to have arbitrary values. Our experiments suggest that the holdout method delivers the best time-accuracy tradeoff and as such we propose $k = 2$ as a default, but we provide the option of changing it to suit

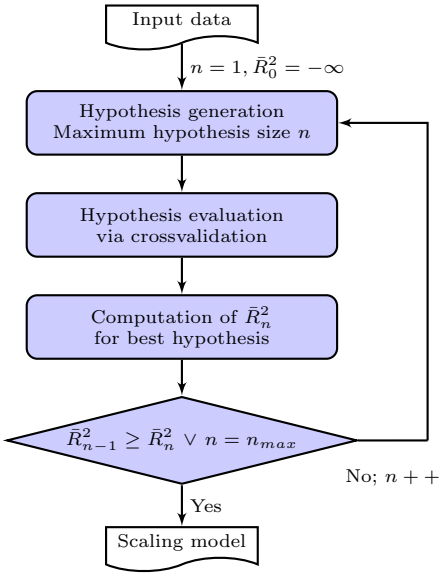


Figure 2: Iterative model refinement process. Solid boxes represent actions or transformations, and banners their inputs and outputs.

the user's needs. When creating the sets we assign adjacent data points to different sets.

## 2.4 Model refinement

To arrive quickly at a suitable model hypothesis and to protect against overfitting, we start with a coarse approximation that we successively refine until we reach the point of *statistical shrinkage* [8]. This is the point after which the model starts to lose predictive power outside the range of samples. The whole refinement process is summarized in Figure 2. At the beginning, we allow a maximum of one term in our hypothesis chosen via cross-validation, as described earlier. We then compute the adjusted coefficient of determination $\bar{R}^2$ [8] of the best model we can find. In the next iteration, we allow a maximum of two terms in the hypothesis. We repeat the cross-validation to find the best model and compute the new adjusted $\bar{R}^2$. If the new value computed for the adjusted $\bar{R}^2$ is smaller than the previous one, indicating that adding more terms to the hypothesis would lead to statistical shrinkage, then the iterative process stops. Otherwise, we continue adding terms to the model until we reach either the shrinkage point or a configurable limit, which may be the maximum time we are prepared to invest or the maximum number of terms we want to consider. Since our method avoids counterproductive refinement and can impose a cap on the iteration depth, we can tackle even larger multi-parameter modeling problems.

## 2.5 Performance extrapolation

Once we have reached this point in our workflow, we have a model describing the scaling behavior of each call path in our application. Now, we can evaluate the scaling function for a target scale $p_t$ or just look at the asymptotic behavior. We can either extrapolate execution time or requirements (e.g., bytes sent/received or floating point operations). The latter can also be helpful in finding roofline models [36], which take resource limitations into account that become effective only at larger scales, an extension of our method which is already in progress. Extrapolating requirements is also relevant to system design because it allows the hardware resources to be optimally balanced according to an application's future needs. Of course, the model we create can only reflect information and phenomena present in the data. As such, any projections will not account for effects that only come into play outside the scope of the experiment. A simple example is the speed-up effect achieved when the data of a strong-scaling application fits completely into the cache. Unless this occurs within the experimental data the method will not predict its effect on performance. Regardless of whether the user chooses a specific target scale $p_t$ or is just interested in the asymptotic behavior, we are now in a position to rank all call paths by their expected performance impact. Those at the top of the list are the kernels whose models the user should compare to his or her expectations and analyze further if serious discrepancies arise.

## 2.6 Kernel refinement

Once the kernels relevant at the target scale have been determined, the user may find the granularity of these kernels too coarse and, as a consequence, the resulting performance model too complex to draw meaningful conclusions. This can happen if the default instrumentation of Scalasca, which is typically applied at the level of functions, is not fine-grained enough to pinpoint pieces of code that are small enough for inspection by a human user. In this case, the instrumentation around the kernels of interest can be narrowed or the kernels split into multiple pieces to be modeled separately. At the same time, the instrumentation around those parts of the program that our analysis classifies as irrelevant can be lifted to reduce instrumentation overhead. Then the whole process starts over again: this time with more targeted measurements that exploit the knowledge gained in the previous iteration.

## 2.7 Effort

The required computational effort consists of two components—running the input experiments and running the model generator. As long as only one model parameter is used, as done throughout this study, the latter takes less than a minute on a single processor and is therefore negligible. The cost of the input experiments can be quantified in relation to experiments at the target scale, which our method helps to avoid. In weak scaling mode, the compute time of a perfectly scaling code in node hours is proportional to the number of processors. Assuming that the number of processors is always a power of two, running experiments at input scales $\{2^0, \ldots, 2^m\}$ together is thus less expensive than a single run at $p_t = 2^{m+1}$. If the code scales poorly or the target scale grows beyond $2^{m+1}$, the amortization factor can increase substantially. Jitter may require more experiments per input scale, but to be conclusive experiments at the target scale would have to be repeated as well.

# 3. INTEGRATION IN SCALASCA

To make the development of our tool as cost-efficient as possible, it relies on a standard performance-measurement infrastructure, only adding the performance analytics described in the previous section on top of this. Specifically,

it has been designed as an extension of Scalasca [12], a well-established open-source toolset that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior. There, the tool is embedded in the performance algebra framework [32], which is a data model and format for parallel performance profiles plus utilities (i.e., so-called operators) for their manipulation.

As input, the tool takes a list of profiles and the target scale. The output is currently textual and includes analytical models plus extrapolated execution time and requirements metrics for each call path. Details of the modeling process are specified in a configuration file that defines parameters such as the number of terms and the exponents to be used in hypothesis creation and so forth. Defaults geared towards single-parameter scalability performance modeling ensure that the tool works out of the box for first-time users interested in finding scalability bugs. To include other model parameters besides the number of processes, users would specify their values for each input run. The number of processes itself do not need to be mentioned explicitly as this information can be extracted from the profile. Of course, the constituent terms of the hypothesis can be different for each model parameter. A major goal of this tool is to provide a powerful "push-button" mechanism that works without assuming prior experience in the field of performance modeling, while providing all relevant levers for control and steering of the process to advanced users who wish to customize it.

# 4. EVALUATION

We illustrate the capabilities of our tool using three MPI applications. Specifically, we demonstrate that our tool

- identifies a scalability issue in a code that is known to have one,
- does not identify a scalability issue in a code that is known to have none, and
- identifies two scalability issues in a code that was thought to have only one.

In the first two cases, we find the models we generate automatically to be in good agreement with manually created models previously reported in the literature. In the third case, we are not aware of any pre-existing performance model. In the second case, we further show that we can produce accurate models for model parameters other than the number of processes.

We performed our experiments on the IBM BlueGene/Q system Juqueen and the Sun cluster Juropa at the Jülich Supercomputing Centre. Juqueen is a large leadership supercomputer with almost 500,000 cores, ranked $7^{th}$ in the TOP500 list as of June 2013. Each node features one PowerPC A2 processor with 16 cores running at 1.6 GHz. Juropa is a compute cluster composed of 2,208 nodes, each equipped with two Intel Xeon X5570 (Nehalem-EP) quad-core processors running at 2.93 GHz. Unless otherwise stated, we always used the default settings for $n$, $I$, $J$ specified in Section 2.3. We ran the model generator on several desktop systems and front-end nodes, where model generation for a single but full code never exceeded one minute.

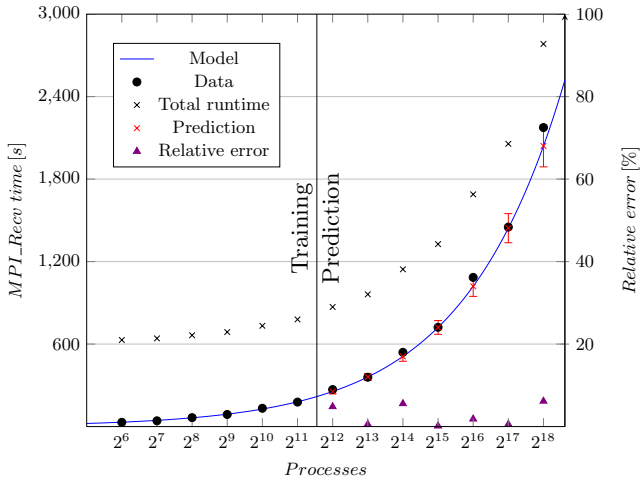


Figure 3: Measured vs. predicted execution time of the two receive operations involved in the wavefront process of Sweep3D on Juqueen.

## 4.1 SWEEP3D

In this example, we show how our tool helps identify and explain a scalability problem, providing a first impression of the user experience. The Sweep3D benchmark [23] is a compact application that solves a 1-group time-independent discrete ordinates neutron transport problem. It was extracted from a real ASCI code. The program calculates the flux of neutrons through a three-dimensional grid along several angles of travel. To partition the problem, the code maps the three-dimensional domain onto a two-dimensional grid of processes. Parallelism is achieved through a pipelined wavefront process that propagates data along diagonal lines through the grid. The particular angle being processed at a given moment determines the direction of the wavefront, which can originate from any of the four grid corners. The pipeline organization enables multiple wavefronts to follow each other along the same direction, although the inability of a process to satisfy horizontal and vertical neighbors at the same time introduces propagating delays. Parallel efficiency drops further whenever the pipeline has to be refilled after the direction has changed. In both cases, the consequences are wait states that materialize in receive operations.

The literature mentions accurate models [19, 35] that describe the performance behavior of wavefront processes as they occur in Sweep3D on various architectures. The LogGP model reported in [19] characterizes the communication time as follows:

$$t^{comm} = [2(p_x + p_y - 2) + 4(n_{sweep} - 1)] \cdot t_{msg} \qquad (2)$$

$p_x$ and $p_y$ denote the lengths of the process-grid edges, $n_{sweep}$ the number of wavefronts to be computed, and $t_{msg}$ the time needed for a one-way nearest-neighbor communication. Given that both $n_{sweep}$ and $t_{msg}$ are largely independent of the number of processes $p$ and that in our experiments $p_x = p_y$ and $p = p_x \cdot p_y$, we can rewrite Equation (2) as:

$$t^{comm} = c \cdot \sqrt{p} \qquad (3)$$

The (combined) model generated by our tool for the two

Table 1: The most time-consuming Sweep3D kernels (i.e., call paths) ranked by their predicted execution time at the target scale $p_t = 262,144$ processes. The values and models reflect exclusive execution times without callees. The predictive error applies only to $p_t$. On the left we used training data with up to 2,048 processes ($p_t = 128 \cdot p_{max}$), on the right with up to 8,192 processes ($p_t = 32 \cdot p_{max}$).

| Kernel | Runtime [%] | Increase | $P_1$ ($p_i \leq 2,048$) | | $P_2$ ($p_i \leq 8,192$) | |
|---|---|---|---|---|---|---|
| | $p_t = 262k$ | $\frac{t(p=262k)}{t(p=64)}$ | Model [s] $t = f(p)$ | Predictive error [%] | Model [s] $t = f(p)$ | Predictive error [%] |
| `sweep` → `MPI_Recv` | 65.35 | 16.54 | $3.99 \cdot \sqrt{p}$ | 6.16 | $4.03 \cdot \sqrt{p}$ | 5.10 |
| `sweep` | 20.87 | 0.23 | 582.19 | 0.01 | 582.19 | 0.01 |
| `global_int_sum` → `MPI_Allreduce` | 12.89 | 18.68 | $0.94\sqrt{p} + 0.04\sqrt{p}\log p$ | 23.00 | $1.06\sqrt{p} + 0.03\sqrt{p}\log p$ | 13.60 |
| `sweep` → `MPI_Send` | 0.40 | 0.23 | 11.66 | 29.00 | $11.49 + 0.09\sqrt{p}\log p$ | 15.40 |
| `source` | 0.25 | 0.04 | $6.86 + 9.68 \cdot 10^{-5}\log p$ | 0.01 | $6.86 + 9.13 \cdot 10^{-5}\log p$ | 0.01 |

receive operations involved in the wavefront process (sweep → MPI_Recv) is $3.99 \cdot \sqrt{p}$ and, thus, consistent with Equation (3). As Figure 3 illustrates, it also matches our measurements on Juqueen quite accurately. The two receive operations are modeled together because Scalasca's default instrumentation merges them into one call path. Note that we do not need large application runs to accurately determine the model. The figure presents results based on only six training and evaluation data points with the process counts $P_1 = \{2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}\}$ and we extrapolate up to 262k processes. The difference between prediction and measurement never exceeds 7%. Using more training and evaluation data points by adding measurements such that $P_2 = P_1 \cup \{2^{12}, 2^{13}\}$, the model becomes even more precise.

That the requirements models for both the number of bytes and the number of messages received predict constant values independent of the number of processes suggests that any increase in communication time is caused by wait states. Because the wavefront travels along the diagonal of the process grid, waiting times proportional to the square root of the number of processes can actually be expected. Having waiting time grow with $\sqrt{p}$ means that every quadrupling of $p$ will double its amount, which can hardly be classified as scalable.

Table 1 lists the five kernels that would consume most of the time at the target scale $p_t = 262,144$ processes, ranked by their predicted execution time. To underline that indeed the right kernels appear at the top, we show their measured execution time in terms of both their relative contribution at $p_t$ and the increase factor of their execution time in comparison to $p = 64$. The latter offers some intuition on how seriously the performance is affected. Together, all five kernels account for more than 99% of the overall runtime. Although our predictions based on training data with up to 8k processes are closer to measured values, even predictions based on training data with up to 2k still show the same general trend. In particular, the ranking remains unchanged. Note that adding more data points does not change the model hypothesis for four of the five kernels, only their coefficients vary slightly to reflect the increased precision allowed by the additional training points. The changing model for $sweep \rightarrow MPI\_Send$ reflects that the new training points manifest a new effect which was previously impossible to see at smaller scales. In this specific case, the runs at 2k processes and beyond allow the latency effect of communication leaving the node board to be observed.

Because the amount of waiting time in Sweep3D, which is responsible for the bulk of the time spent in MPI at larger scales, depends on the progress of the wavefront computation, earlier studies [19, 35] concluded that single-node performance is the most serious impediment to the scalability of Sweep3D—and not, for example, the saturation of network resources. To see whether we arrive at the same conclusion using our automated approach, we also conducted experiments on Juropa, whose cores are much more powerful than Juqueen's. Results for the two platforms obtained with training data from runs with up to 2,048 processes are again consistent with manually developed models. While the sweep() routine, where the wavefront computation takes place, is about eight times faster on Juropa, the receive inside, where the wait states accumulate, is eight times slower on Juqueen. Otherwise, the models we generate for the two kernels on Juqueen and Juropa are identical.

On a final note, this relationship sheds light also on a performance phenomenon observed in a more recent experimental study of Sweep3D [38], which analyzes the consequences of load imbalance between a central rectangular region and the rest of the process grid, which is caused by a corrective function invoked only during certain iterations. Since overload has effects similar to processors with lower speed, it is likely to enlarge only the coefficient of the $\sqrt{p}$ term in the model of the dominant receives and, thus, to have only little bearing on the general scalability. Applying our tool to the affected iterations only, we found this coefficient to be enlarged by 20% but otherwise observed the same scaling behavior.

## 4.2 MILC

In this case study, we show that our tool characterizes also a scalable application correctly. In addition, we show how our tool derives time and requirements models for model parameters beyond the number of processes. MILC is a set of codes written in C for studying quantum chromodynamics (QCD)

Table 2: Automatically generated models of selected functions in MILC when varying the number of processes. The prediction errors were computed with resepct to a target scale of 65,536 processes.

| Kernel | Model [s] $t = f(p)$ | $\lvert 1 - R^2 \rvert$ | Predictive error [%] |
|---|---|---|---|
| `CGSF` | 0.024 | 0.000 | 0.43 |
| `MPI_Allreduce` | $6.30 \cdot 10^{-6} \cdot \log^2 p$ | 0.084 | 12.77 |
| `MASFL` | 0.004 | 0.000 | 0.04 |

Table 3: Automatically generated models of selected functions in MILC when varying the number of grid points per process. For the underlying experiments, we used the following parameters: `meas=5`, `warms=0`, `trajecs=1`, `traj_between_meas=1`, `steps_per_trajectory=10`.

| Kernel | Flops | | Invocations | | Flops/invocation | |
|---|---|---|---|---|---|---|
| | Model | $\lvert 1-R^2\rvert$ | Model | $\lvert 1-R^2\rvert$ | Model | $\lvert 1-R^2\rvert$ |
| | $flops = f(V)$ | $[\cdot 10^{-3}]$ | $invocations = f(V)$ | $[\cdot 10^{-3}]$ | $\frac{flops}{invoc.} = f(V)$ | $[\cdot 10^{-3}]$ |
| `load_lnglinks` | $5.64 \cdot 10^4 \cdot V$ | 0.030 | $2.31 \cdot 10^3$ | 0.000 | $24.42 \cdot V$ | 0.030 |
| `load_fatlinks_cpu` | $1.95 \cdot 10^6 \cdot V$ | 0.210 | $7.14 \cdot 10^4$ | 0.000 | $27.36 \cdot V$ | 0.210 |
| `ks_congrad` | $1.16 \cdot 10^8 + 3.24 \cdot 10^5 \cdot V^{\frac{5}{4}}$ | 0.292 | $5.11 \cdot 10^4 + 1.38 \cdot 10^4 \cdot V^{\frac{1}{4}}$ | 4.000 | $15.94 \cdot V$ | 0.143 |
| `imp_gauge_force_cpu` | $1.65 \cdot 10^6 \cdot V$ | 0.015 | $7.40 \cdot 10^4$ | 0.000 | $22.28 \cdot V$ | 0.015 |
| `eo_fermion_force_twoterms_site` | $4.02 \cdot 10^6 \cdot V$ | 0.002 | $1.27 \cdot 10^5$ | 0.000 | $31.61 \cdot V$ | 0.002 |

via parallel simulations of the SU(3) lattice gauge theory on a four-dimensional lattice. In earlier work [16], analytical models were manually created that describe the behavior of MILC/su3_rmd, one of the MILC codes, by characterizing its most important components with respect to a number of parameters. We now show that our modeling tool chain allows similar models to be derived automatically.

We first consider weak scaling runs on Juqueen, increasing the number of processes linearly with the problem size. The existing models suggest that MILC is highly scalable code, that is, the time per process should remain constant except for a rather small logarithmic term caused by global convergence checks. As we show below, our method correctly determines the most important features of this model. Specifically, we demonstrate the tool's ability to derive scalability models for the execution time of three representative kernels: *compute_gen_staple_field*, *g_vecdoublesum* $\rightarrow$ *MPI_Allreduce*, and *mult_adj_su3_fieldlink_lathwvec*, which we abbreviate as CGSF, MPI_Allreduce, and MASFL, respectively. Given that MILC is known to scale well, we refined the default setting for $I$ by adding $\left\{\frac{1}{4}, \frac{1}{3}, \frac{2}{3}, \frac{3}{4}\right\}$, as suggested in Section 2.3. We collected five data points for each function at the scales $P_3 = \left\{2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}\right\}$ with a local lattice size of $V = 9^4$ per process. All model functions generated for Juqueen are shown in Table 2. We use the residual sum of squares, a quality-of-fit metric, as an indicator for the quality (and thus confidence) of our model. For a fit of $n$ variables with measurement values $y_i$ and fitted hypothesis model $f(x_i)$ $(1 \leq i < n)$, $RSS = \sum_{i=1}^{n}(y_i - f(x_i))^2$. We calculate the coefficient of determination $R^2 = 1 - \frac{RSS}{TSS}$ as a measure of fit, where $TSS = \sum_{i=1}^{n}(y_i - \bar{y})^2$. If $R^2 = 1$, the model fits the data exactly. For ease of understanding, we show $|1 - R^2|$, the absolute difference between $R^2$ and the optimum, which can be considered a normalized error, in the table.

Beyond scalability in terms of the number of processes, we also derive requirements models for the size and number of MPI point-to-point messages as a function of grid points per process. This demonstrates the ability of our tool to generate models for different input parameters—in this example to predict the effects of different process-local grid sizes. For four different performance-critical kernels, the handcrafted model characterizes the message size as $18s\sqrt[4]{V^3}$ bytes, with $s$ being the size of a floating-point value in bytes and $V$ being the number of grid points per process. Our input measurements, which we took on Juropa with its more generous memory per node, were made with a fixed number of processes $p = 32$, single precision ($s = 4$ bytes), and a varying number of grid points $\mathcal{V} = \{81, 256, 400, 625, 900, 1080, 1296, 1512, 1764, 2058, 2401\}$. Since there is no performance variation in these requirements measurements, the quality of the automated fit (and thus the confidence) is high, resulting in a model that matches the handcrafted counterpart exactly. Our method also found the number of messages in each kernel to be invariant regardless of the lattice size, which further matches the models in [16]. Another metric analyzed was the number of floating-point operations in each invocation of the time-intensive kernels as a function of the number of grid points per process. The results in Table 3 show that the number of floating-point operations per kernel invocation is proportional to the number of grid points (rightmost column), which is again consistent with [16]. All kernels but the conjugate-gradient kernel (`ks_congrad`) have a constant number of invocations, whereas the number of times the conjugate-gradient kernel is invoked depends for this particular input matrix on the number of grid points (middle column).

In summary, our method was able to reproduce the most significant parts of the models that were manually created to describe the behavior of MILC as a function of the number of processes or the local volume. The requirement models for the number of messages, their sizes, and the number of floating-point operations per lattice point can be very useful for architecture co-design. We did not show additional models for cache misses and other metrics because they follow the same principle. Our results demonstrate that automation can lead to good performance models with low manual effort.

## 4.3 HOMME

To showcase how our tool helps to find hidden scalability bugs in a production code for which no performance model was available, we applied it to HOMME [11], the dynamical core of the Community Atmospheric Model (CAM) being developed at the National Center for Atmospheric Research (NCAR). HOMME, which was designed with scalability in mind, employs spectral element and discontinuous Galerkin methods on a cubed sphere tiled with quadrilateral elements. While experiences in the past did not indicate any scalability issues with up to 100,000 processes, HOMME was never subjected to a systematic scalability study. All the results we

Table 4: Models of the kernels of HOMME derived from smaller and larger-scale input configurations. The predictive error refers to the target scale of $p_t = 130k$.

| Kernel | $P_4(p_i \leq 15,000)$ | | $P_5(p_i \leq 43,350)$ | |
|---|---|---|---|---|
| | Model [s] $t = f(p)$ | Predictive error [%] | Model [s] $t = f(p)$ | Predictive error [%] |
| `box_rearrange` → `MPI_Reduce` | $0.026 + 2.53 \cdot 10^{-6} \cdot p\sqrt{p} + 1.24 \cdot 10^{-12} \cdot p^3$ | 57.02 | $3.63 \cdot 10^{-6} \cdot p\sqrt{p} + 7.21 \cdot 10^{-13} \cdot p^3$ | 30.34 |
| `vlaplace_sphere_wk` | 49.53 | 99.32 | $24.44 + 2.26 \cdot 10^{-7} \cdot p^2$ | 4.28 |
| `laplace_sphere_wk` | 44.08 | 99.32 | $21.84 + 1.96 \cdot 10^{-7} \cdot p^2$ | 2.34 |
| `biharmonic_wk` | 34.40 | 99.33 | $17.92 + 1.57 \cdot 10^{-7} \cdot p^2$ | 3.43 |
| `divergence_sphere_wk` | 16.88 | 99.31 | $8.02 + 7.56 \cdot 10^{-8} \cdot p^2$ | 4.25 |
| `vorticity_sphere` | 9.74 | 99.55 | $6.51 + 7.09 \cdot 10^{-8} \cdot p^2$ | 8.66 |
| `divergence_sphere` | 15.36 | 99.33 | $7.74 + 6.91 \cdot 10^{-8} \cdot p^2$ | 0.95 |
| `gradient_sphere` | 14.77 | 99.33 | $6.33 + 6.88 \cdot 10^{-8} \cdot p^2$ | 5.17 |
| `advance_hypervis` | 9.76 | 99.25 | $5.5 + 3.91 \cdot 10^{-8} \cdot p^2$ | 1.47 |
| `compute_and_apply_rhs` | 48.68 | 1.65 | 49.09 | 0.83 |
| `euler_step` | 28.08 | 0.51 | 28.13 | 0.33 |

present here for this code reflect measurements on Juqueen based on an input configuration suggested by the application developer team.

Table 4 lists different kernels of the code, ordered by their asymptotic runtime ($p_t \to \infty$). It shows the models produced for two different sets of input configurations. The first one includes data points at the scales $P_4 = \{600, 1176, 4056, 7776, 13824, 14406, 15000\}$, the second one $P_5 = P_4 \cup \{15606, 16224, 23814, 31974, 43350\}$ adds more measurements to the initial set. The order in the table is based on models determined using $P_5$. The models derived from $P_4$ show constant runtimes for all kernels except for the reduce in *box_rearrange*, which grows with $p^3$. Deriving the models from the larger set introduces a dependence on $p^2$ (with a small factor) for all but one of the hitherto constant kernels. Obviously, the enlarged set reveals a phenomenon not visible in the smaller set. If the number of processes is large enough, both the quadratic and the cubic terms will turn into serious bottlenecks, contradicting our initial expectation the code would scale well. The table also shows the predictive error, which characterizes the deviation of the prediction from measurement at the taget scale $p_t = 130k$, highlighting the benefits of including the extra data points.

After looking at the number of times any of the quadratic kernels was invoked at runtime, a metric we also measure and model, the quadratic growth was found to be the consequence of an increasing number of iterations inside a particular subroutine. Interestingly, the formula by which the number of iterations is computed contained a *ceiling* term that limits the number of iterations to one for up to and including 15k processes. Beyond this threshold, a term depending quadratically on the process count causes the number of iterations executed to grow rapidly, causing a significant drop in performance. It turned out that the developers were aware of this issue and had already developed a temporary solution, involving manual adjustments of their production code configurations. Specifically, they fix the number of iterations and carefully tune other configuration parameters to ensure numerical stability. Nevertheless, the issue was correctly detected by our tool. Given the tuning necessary to ensure numerical stability, a weak scaling analysis of the workaround is beyond the scope of this paper.

In contrast to the previous problem, the cubic growth of the time spent in the reduce function was previously unknown. The reduction is needed to funnel data to dedicated I/O processes. The coefficient of the dominant term at scale is very small (i.e., in the order of $10^{-13}$). While not being visible at smaller scales, it will have an explosive effect on performance at larger scales, becoming significant even if executed just once. The reason why this phenomenon remained unnoticed until today is that it belongs to the initialization phase of the code that was not assumed to be performance relevant in larger production runs. While still not yet crippling in terms of the overall runtime, which is in the order of days for production runs, the issue already cost more than one hour in the large-scale experiments we conducted. The problem was reported to the developers at NCAR, who are currently working on a solution. The example demonstrates the advantage of modeling the entire application instead of only selected candidate kernels expected to be time intensive. Some problems might simply escape attention because non-linear relationships make our intuition less reliable at larger scales. Note that coefficients such as $10^{-13}$ are small in view of the typical run-to-run deviation, but have to be seen in relation to the associated polynomial expression $p^3$, which became larger than $10^{12}$ in our input experiments. Given that the target scale is usually one or more orders of magnitude greater than the largest input scale, fully accurate coefficients are therefore secondary when trying to locate scalability bottlenecks with higher exponents of $p$. Moreover, within the scope of our approach the coefficients of scalable model functions with lower exponents of $p$ are of minor interest anyway.

Figure 4 summarizes our two findings and compares our predictions with actual measurements. While the quadratically growing iteration count seems to be more urgent now, the reduce might become the more serious issue in the future.

# 5. RELATED WORK

Analytical performance modeling techniques have been used to model the performance of numerous important applications manually [21, 25]. It is well understood that analytical models have the potential of providing important insights into complex behaviors [30]. Performance models also offer insight into different parts of the system. For example, Boyd et al. used performance models to assess the quality of a tool

chain, such as a compiler or runtime system [6]. A very important motivation for the use of performance models was presented by Petrini et al. [28]. In their study, the difference between actual and predicted performance led to the discovery of system noise as the source of seriously degraded performance. In general, there is consensus that performance modeling is a powerful tool for assessing an application's resource consumption and scalability.

Hoefler et al. aimed to further popularize performance modeling by defining a simple six-step process to create application performance models [16]. The described method leads to insight into application scaling behavior but is tedious to apply to real codes and has not yet been explicitly used to predict the scaling behavior of applications. Bauer, Gottlieb, and Hoefler show how to model performance variations in this framework using simple statistical tools [3]. They also describe how to measure the influence of certain system parameters such as the network topology.

Other approaches focus less on human-readable general-purpose models but rather on models generated for a very specific purpose. For example, Ipek et al. propose multilayer artificial neural networks to *learn* application performance [20] and Lee et al. compare a set of different schemes for automated machine-based performance learning and prediction [22]. Zhai, Chen, and Zheng extrapolate single-node performance to complex parallel machines [40]. Wu and Müller [37] extrapolate traces to larger process counts and can thus predict communication operations. Their extrapolation relies on a trace compression scheme that assumes regular communications. Our method is based on lightweight profiles which can be generated without making prior assumptions. All these schemes aim to deliver the most accurate prediction but do not try to find the simplest human-readable scaling function, thus limiting insight.

A second objective of performance modeling is to predict application performance on a different target architecture.

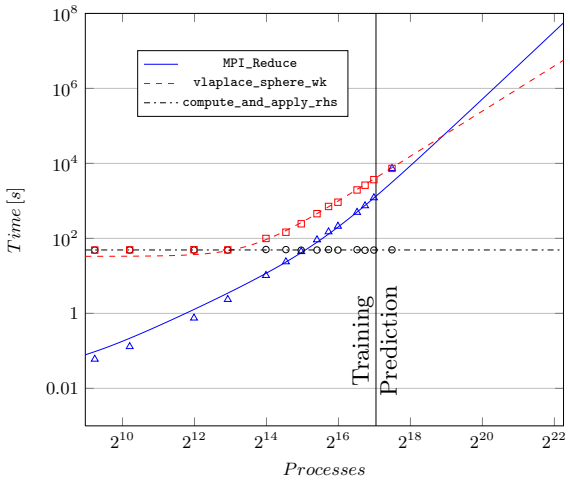


Figure 4: Runtime of selected kernels in HOMME as a function of the number of processes. The graph compares predictions (dashed or contiguous lines) to measurements (small triangles, squares, and circles).

Carrington et al. propose a model-based prediction framework for applications on different computers [7], Marin and Mellor-Crummey demonstrate how application models can be derived semi-automatically to predict performance on different architectures [24], and Yang, Ma, and Müller model application performance on different architectures by running kernels on the target architecture [39].

A related line of work uses simulation to predict application performance on different systems. Simulators range from cycle accurate [5, 31] to abstract model-driven message passing simulation [18]. Trace-driven simulators such as SimGrid [9], DIMEMAS [33], and PSINS [34] simulate more detailed network models. Other simulators, such as BigSim [42], Silas [15], and MPI-SIM [1], use direct or kernel execution to assess computation or communication times more accurately. However, direct execution approaches often have prohibitive memory requirements [26]. As opposed to our semi-analytic modeling method, simulations require huge resources for their execution and deliver little insight into scaling behavior on their own. However, they could be integrated into our method as a way to generate larger output predictions.

The PACE toolset [27] provides performance modeling features but the techniques are not described in detail and the tool was not available to us. We argue that our approach is much simpler to implement in a tool. Coarfa et al. automatically compare pairs of measurements at different scales to identify scalability bottlenecks [10], whereas our approach creates explicit predictive models that describe the scaling behavior beyond the range of measurements. Barnes et al. use regression analysis to predict the scalability of applications [2] and is probably the most similar work. The main differences are that they aim to predict the optimal number of CPUs to solve a certain problem while we are most interested in predicting the CPU time consumed for a specific run. For this, their tool considers strong scaling of the whole application while we focus on identifying non-scalable functions in the code. In addition, Barnes et al. assume a load-balanced application for their run while we are able to detect the scalability limitations caused by load imbalance.

# 6. CONCLUSION

Our results confirm that automated performance modeling is feasible and that the automatically generated models are accurate enough to identify scalability bugs. In fact, in those cases where hand-crafted models existed in the literature we found our models to be competitive. The main lesson that we learned during our work is that the advantages of mass production also apply to performance models. First, approximate models are acceptable as long as the effort to create them is low and they do not mislead the user. Second, code coverage is as important as model accuracy. Having approximate models for all parts of the code can be more useful than having a model with 100% accuracy for just a tiny portion of the code or no model at all. Extending this argument beyond the boundaries of a single application, we believe that our tool will make scalability modeling accessible to a much wider audience of HPC developers and applications.

Our tool models only behaviors found in the training data. We provide direct feedback information regarding the num-

ber of runs required to ensure statistical significance of the modeling process itself, but there is no automatic way of determining at what scale particular behaviors start manifesting themselves. In the HOMME example, the iteration count suddenly increased after 15k processes, which was only detectable through either code analysis or experiments. We expect that our method will be most effective for regular problems with repetitive behavior, whereas irregular problems with strong and potentially non-deterministic dynamic effects will require enhancements of our method.

The models we generate are primarily designed to allow projection for scaled-up versions of the current system. However, the requirements models we create alongside the execution-time models are largely independent of a specific hardware architecture. If combined with measured or assumed hardware characteristics, they can be used to make projections for other existing systems and can suggest how a system must be designed to maximize the performance of a given code.

In the future, we plan to use our tool to study the influence and value of additional hardware information such as cache size, network bandwidth and latency. Incorporating such additional constraints into the modeling process may improve precision and help in combination with requirements models to accurately predict jumps and plateaus caused by the saturation of such resources, allowing the formulation of roofline models. We are also looking into heuristics for traversing the hypothesis search space more efficiently, which would improve the modeling speed further and allow the number of model parameters to be increased.

*Acknowledgments.* This work was supported by the German Research Foundation (DFG) and the Swiss National Science Foundation (SNSF) through the German Priority Programme 1648 *Software for Exascale Computing* (SPPEXA). Further support received from the G8 Research Councils Initiative on Multilateral Research, Interdisciplinary Program on Application Software towards Exascale Computing for Global Scale Issues is gratefully acknowledged. Finally, we would like to thank John Dennis and Rich Loft from the National Center for Atmospheric Research, Boulder, Colorado, USA, for giving us access to HOMME and for their assistance in running our experiments.

## 7. REFERENCES

[1] R. Bagrodia, E. Deelman, and T. Phan. Parallel simulation of large-scale parallel applications. *Intl. Journal of High Performance Computing Applications*, 15(1):3–12, 2001.

[2] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proc. of the Intl. Conference on Supercomputing*, (ICS), pages 368–377. ACM, 2008.

[3] G. Bauer, S. Gottlieb, and T. Hoefler. Performance modeling and comparative analysis of the MILC lattice QCD application su3_rmd. In *Proc. of CCGrid*, May 2012.

[4] A. Blum, A. Kalai, and J. Langford. Beating the hold-out: bounds for k-fold and progressive cross-validation. In *Proc. of the 12th Annual Conference on Computational Learning Theory*, (COLT), pages 203–208. ACM, 1999.

[5] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Performance Eval. Review*, 31:8–12, March 2004.

[6] E. L. Boyd, W. Azeem, H.-H. Lee, T.-P. Shih, S.-H. Hung, and E. S. Davidson. A hierarchical approach to modeling and improving the performance of scientific applications on the ksr1. In *Proc. of the Intl. Conference on Parallel Processing*, (ICPP), pages 188–192. IEEE Computer Society, 1994.

[7] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, February 2006.

[8] D. S. Carter. Comparison of different shrinkage formulas in estimating population multiple correlation coefficients. *Educational and Psychological Measurement*, 39(2):261–266, 1979.

[9] H. Casanova, A. Legrand, and M. Quinson. Simgrid: a generic framework for large-scale distributed experiments. In *Proc. of the 10th Intl. Conference on Computer Modeling and Simulation*, (UKSIM), pages 126–131. IEEE Computer Society, 2008.

[10] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of SPMD codes using expectations. In *Proc. of the 21st Intl. Conference on Supercomputing*, (ICS), pages 13–22. ACM, 2007.

[11] J. M. Dennis, J. Edwards, K. J. Evans, O. Guba, P. H. Lauritzen, A. A. Mirin, A. St-Cyr, M. A. Taylor, and P. H. Worley. CAM-SE: A scalable spectral element dynamical core for the community atmosphere model. *Intl. Journal of High Performance Computing Applications*, 26(1):74–89, 2012.

[12] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[13] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(3):12–21, 1993.

[14] D. M. Hawkins, S. C. Basak, and D. Mills. Assessing model fit by cross-validation. *Journal of Chemical Information and Computer Sciences*, 43(2):579–586, 2003.

[15] M.-A. Hermanns, M. Geimer, F. Wolf, and B. J. N. Wylie. Verifying causality between distant performance phenomena in large-scale MPI applications. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 78–84. IEEE Computer Society, February 2009.

[16] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, SC '11, pages 6:1–6:12. ACM, 2011.

[17] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC '10)*, November 2010.

[18] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim: simulating large-scale applications in the LogGOPS model. In *Proc. of the 19th ACM Intl. Symposium on High Performance Distributed Computing*, (HPDC), pages 597–604. ACM, 2010.

[19] A. Hoisie, O. M. Lubeck, and H. J. Wasserman. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Workshop on Wide Area Networks and High Performance Computing*, pages 171–187. Springer-Verlag, 1999.

[20] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Proc. of the 11th Intl. Euro-Par Conference*, pages 196–205. Springer-Verlag, 2005.

[21] 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)*, page 37. ACM, 2001.

[22] 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 *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP '07), pages 249–258. ACM, 2007.

[23] Los Alamos National Laboratory. ASCI SWEEP3D v2.2b: Three-dimensional discrete ordinates neutron transport benchmark, 1995.

[24] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Performance Eval. Review*, 32(1):2–13, June 2004.

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

[26] C. Mei. A preliminary investigation of emulating applications that use petabytes of memory on petascale machines. Master's thesis, University of Illinois at Urbana-Champaign, 2007.

[27] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace–a toolset for the performance prediction of parallel and distributed systems. *Intl. Journal of High Performance Computing Applications*, 14(3):228–251, August 2000.

[28] 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*, (SC '03), page 55. ACM, 2003.

[29] R. R. Picard and R. D. Cook. Cross-validation of regression models. *Journal of the American Statistical Association*, 79(387):575–583, 1984.

[30] 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)*, pages 279–284, 2007.

[31] A. F. Rodrigues, R. C. Murphy, P. Kogge, and K. D. Underwood. The structural simulation toolkit: exploring novel architectures. In *Proc. of the ACM/IEEE Conference on Supercomputing*, (SC '06). ACM, 2006.

[32] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proc. of the Intl. Conference on Parallel Processing (ICPP)*, pages 63–72, Montreal, Canada, August 2004. IEEE Society.

[33] V. Subotic, J. C. Sancho, J. Labarta, and M. Valero. A simulation framework to automatically analyze the communication-computation overlap in scientific applications. In *Proc. of the IEEE Conference on Cluster Computing*, (Cluster '10), pages 275–283. IEEE Computer Society, 2010.

[34] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely. PSINS: An open source event tracer and execution simulator for MPI applications. In *Proc. of the Euro-Par Conference*, pages 135–148. Springer-Verlag, 2009.

[35] H. Wasserman, A. Hoisie, O. Lubeck, and O. Lubeck. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The Intl. Journal of High Performance Computing Applications*, 14:330–346, 2000.

[36] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[37] X. Wu and F. Müller. Scalaextrap: Trace-based communication extrapolation for spmd programs. *ACM Transactions on Programming Languages and Systems*, 34(1), April 2012.

[38] B. J. N. Wylie, M. Geimer, B. Mohr, D. Boehme, Z. Szebenyi, and F. Wolf. Large-scale performance analysis of SWEEP3D with the Scalasca toolset. *Parallel Processing Letters*, 20(04):397–414, 2010.

[39] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proc. of the ACM/IEEE Conference on Supercomputing*, (SC '05), page 40. IEEE Computer Society, 2005.

[40] J. Zhai, W. Chen, and W. Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. *SIGPLAN Notices*, 45(5):305–314, January 2010.

[41] P. Zhang. Model selection via multifold cross validation. *The Annals of Statistics*, 21(1):pp. 299–313, 1993.

[42] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *Proc. of the 18th Intl. Parallel and Distributed Processing Symposium (IPDPS)*, page 78, April 2004.