Mass-producing insightful performance models

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

Felix Wolf
German Research School
for Simulation Sciences
RWTH Aachen University
Aachen, Germany
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 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. By automatically generating empirical performance models for each function in the program, we make this powerful methodology easier to use and expand its coverage. This article gives an overview of the method and assesses its potential.**

I. 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, but often also major code surgery in the aftermath. 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.

II. Automated Performance Modeling

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. Our method can be applied to both strong scaling and weak scaling runs. 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 not only execution time but also requirements, our results can 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, we focus on message passing programs. For a detailed description, the reader may refer to [1].

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. We call these regions *kernels* because they define the code granularity at which we generate our models. The output of our tool is a list of program regions, ranked either by their predicted execution time at a target scale of $p_t > p_{max}$ processors or by their asymptotic behavior. Figure 1 gives an overview of the different steps necessary to find scalability bugs. To ensure a statistically relevant set of performance data, profile measurements may have to be repeated several times. 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. Finally, if the granularity of our program regions is not sufficient to arrive at an actionable recommendation, the kernels under investigation can be further refined via more detailed instrumentation.

A. Model generation

Model generation forms the core of our method. 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):

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

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 respectively are chosen from, 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

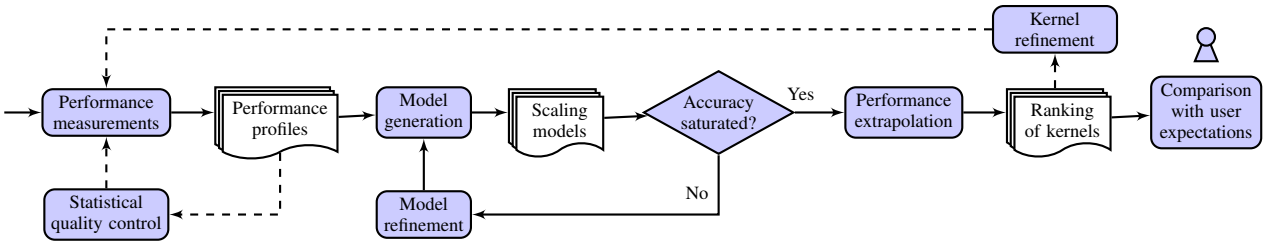

Fig. 1: Performance modeling workflow. Solid boxes represent actions or transformations, and banners their inputs and outputs. Dashed arrows indicate optional paths taken after user decisions.

with the best fit. Then we apply cross-validation [2] to select the hypothesis with the best fit across all candidates.

III. Related Work

Analytical performance modeling has a long history. Early manual models showed to be very effective to describe many qualities and characteristics of applications, systems, and even entire tool chains [3], [4], [5], [6], [7]. Hoefler et al. established a simple six-step process to guide manual performance modeling [8], which served as a blueprint for our automated workflow. Assertions and source-code annotations support developers in the creation of analytical performance models [9], [10], [11].

Various automated modeling methods exist. Many of these tools focus on learning the performance characteristics automatically using various machine-learning approaches [12], [13]. Zhai, Chen, and Zheng extrapolate single-node performance to complex parallel machines using a trace-driven network simulator [14] and Wu and Müller extrapolate traces to predict communications at larger scale [15]. Carrington et al. choose a model from a set of canonical functions to extrapolate traces of applications at scale [16].

IV. Assessment

Challenges addressed. If developers decide to model the scalability of their code today, 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. We are developing 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 and insight of the scalability check but also its speed.

Maturity. We analyzed real-world applications such as climate codes, quantum chromodynamics, fluid dynamics, simulation of the brain and more. We were able to identify the most likely bottlenecks in eight real-world applications, totaling over 10,000 kernels and over 200,000 lines of code.

Uniqueness. The main advantage of the approach lies in the ability to obtain human readable models for each part of the application using only a standard profiling infrastructure and without the need to generate traces. The extremely useful insights that can be gained for comparatively low effort make our approach unique.

Novelty. The novelty of the approach stems from the automation of the modeling process using statistical methods, allowing entire applications to be modeled at a very fine level of granularity. We leverage the following assumptions to achieve our goals:

1) The space of the function classes underlying performance 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*.
3) We create requirements models alongside execution-time models. Their comparison can give a clue to the nature of a scalability problem. Moreover, requirements models are often easier to obtain and well suited for extreme-scale projections.

Applicability. Our approach can be used to model the effect of any parameter deemed important to the behavior of an application, such as the problem size, the number of processes, or the accuracy of the solution. Time is not the only dimension that is modeled. Requirements such as the number of floating point operations, memory accesses, number and size of messages sent are also modeled.

Effort. We already have a working prototype implementation of our method. We plan to expand its capabilities and release it to the public in the near future. 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.

V. Conclusion

The lightweight tool we created can be used to generate useful scalability models for arbitrarily complex codes. Tests on a range of applications confirmed models reported in the literature in cases where such models existed, but also helped uncover a number of previously unknown scalability issues in other cases. Work is underway refining the process to find a good balance between what can be realistically measured while still allowing useful conclusions.

REFERENCES

[1] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC '13)*, 2013, p. 45. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503277

[2] R. R. Picard and R. D. Cook, "Cross-validation of regression models," *Journal of the American Statistical Association*, vol. 79, no. 387, pp. 575–583, 1984. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/01621459.1984.10478083

[3] 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)*, 2001, p. 37. [Online]. Available: http://doi.acm.org/10.1145/582034.582071

[4] M. M. Mathis, N. M. Amato, and M. L. Adams, "A general performance model for parallel sweeps on orthogonal grids for particle transport calculations," College Station, TX, USA, Tech. Rep., 2000. [Online]. Available: http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai%3Ancstrlh%3Atamucs%3Ancstrl.tamu_cs%2F%2FTR00-004

[5] S. Pllana, I. Brandic, and S. Benkner, "Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art," in *Proc. of the 1st Intl. Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2007, pp. 279–284. [Online]. Available: http://dx.doi.org/10.1109/CISIS.2007.49

[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)*, 1994, pp. 188–192. [Online]. Available: http://dx.doi.org/10.1109/ICPP.1994.30

[7] 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)*, 2003, p. 55. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050204

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

[9] N. R. Tallent and A. Hoisie, "Palm: easing the burden of analytical performance modeling," in *Proc. of the International Conference on Supercomputing (ICS)*, 2014, pp. 221–230. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597683

[10] K. Spafford and J. S. Vetter, "Aspen: a domain specific language for performance modeling," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC '12)*, 2012, p. 84. [Online]. Available: http://dl.acm.org/citation.cfm?id=2389110

[11] J. S. Vetter and P. H. Worley, "Asserting performance expectations," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC '02)*, 2002, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/762761.762809

[12] 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*, 2005, pp. 196–205. [Online]. Available: http://dx.doi.org/10.1007/11549468_24

[13] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*, 2007, pp. 249–258. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229479

[14] J. Zhai, W. Chen, and W. Zheng, "Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node," *SIGPLAN Notices*, vol. 45, no. 5, pp. 305–314, 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693493

[15] X. Wu and F. Mueller, "ScalaExtrap: trace-based communication extrapolation for SPMD programs," in *Proc. of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11)*, 2011, pp. 113–122. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941569

[16] L. Carrington, M. Laurenzano, and A. Tiwari, "Characterizing large-scale HPC applications through trace extrapolation," *Parallel Processing Letters*, vol. 23, no. 4, 2013. [Online]. Available: http://dx.doi.org/10.1142/S0129626413400082