Catwalk: A Quick Development Path for Performance Models

Felix Wolf[1,2], Christian Bischof[3], Torsten Hoefler[4], Bernd Mohr[5], Gabriel Wittum[6], Alexandru Calotoiu[1,2], Christian Iwainsky[1], Alexandre Strube[5], and Andreas Vogel[6]

[1] German Research School for Simulation Sciences, 52062 Aachen, Germany
[2] RWTH Aachen University, 52056 Aachen, Germany
[3] Technische Universität Darmstadt, 64293 Darmstadt, Germany
[4] ETH Zurich, CH-8092 Zürich, Switzerland
[5] Forschungszentrum Jülich, 52425 Jülich, Germany
[6] Goethe Universität Frankfurt, 60325 Frankfurt am Main, Germany

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. The objective of the Catwalk project, which is carried out as part of the DFG Priority Programme 1648 Software for Exascale Computing (SPPEXA), is to automate key activities of the performance modeling process, making this powerful methodology easier to use and expanding its coverage. This article gives an overview of the project objectives, describes the results achieved so far, and outlines future work.

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

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 589–600, 2014.

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 the Catwalk project, which is part of the DFG Priority Programme 1648 Software for Exascale Computing (SPPEXA), 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 of the scalability check but also its speed.

The remainder of the paper is structured as follows. Section 2 describes the empirical performance modeling tool and its applications. Section 3 explains the automatic workflow manager used to run the experiments needed as input for the tool. Section 4 outlines ongoing work of extending the current MPI-centric approach towards OpenMP and hybrid applications. One of the target codes for performance modeling, the library UG4, is discussed in Section 5. Finally, we summarize our results and outline future work in Section 6.

2 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 applications. In addition to searching for performance bugs, the models our tool produces also support projections that can be helpful when applying

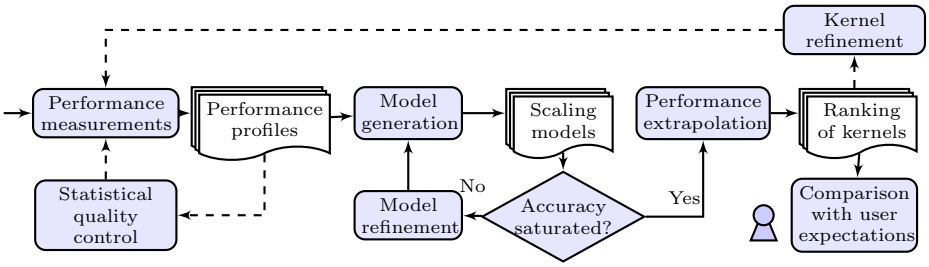

Fig. 1. Workflow of scalability-bug detection. Solid boxes represent actions or transformations, and banners their inputs and outputs. Dashed arrows indicate optional paths taken after user decisions.

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, we focus on message passing programs. For a detailed description including modeling results, the reader may refer to [3].

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 execution of these experiments is supported by a workflow manager, which is described in Section 3. 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.

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. A scalability bug can be any region with a model worse than a given threshold, such as anything scaling worse than linearly. Alternatively, a user can compare the model of a kernel with his own expectations to determine if the performance is worse than expected. 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 Related Work

Analytical performance modeling has a long history. Early manual models showed to be very effective in describing many qualities and characteristics of applications, systems, and even entire tool chains [2, 9, 13, 16, 18]. Hoefler et al. established a simple six-step process to guide manual performance modeling [6], which served as a blueprint for our automated workflow. Assertions and source-code annotations support developers in the creation of analytical performance models [20–22].

Various automated modeling methods exist. Many of these tools focus on learning the performance characteristics automatically using various machine-learning approaches [8, 12]. Zhai, Chen, and Zheng extrapolate single-node performance to complex parallel machines using a trace-driven network simulator [25]. Wu and Müller extrapolate traces to predict communications at larger scale [24]. Carrington et al. choose a model from a set of canonical functions to extrapolate traces of applications at scale [4].

2.2 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):

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

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 [17] to select the hypothesis with the best fit across all candidates. In our experiments we use $I = \{0, 0.5, 1, 1.5, 2, 2.5, 3\}$, $J = \{0, 1, 2\}$ and $n = 5$, and we have observed that it is more than sufficient to accurately represent behaviors found in real world applications.

2.3 Evaluation Summary

We analyzed real-world applications such as climate codes, quantum chromodynamics, fluid dynamics and more. We were able to identify a scalability issue in

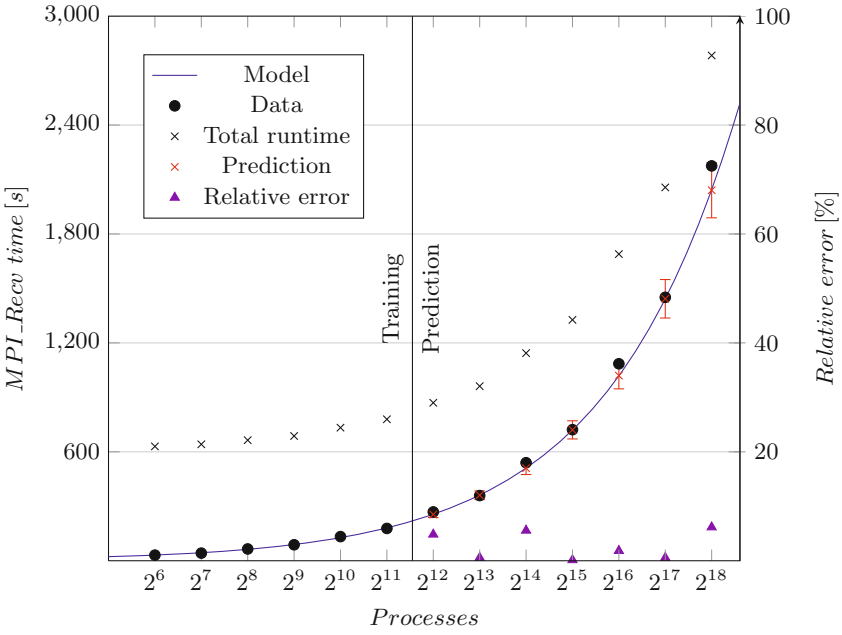

Fig. 2. Measured vs. predicted execution time of the two receive operations involved in the wavefront process of Sweep3D on Juqueen

codes that are known to have such issues (Sweep3D, XNS) and not identify any scalability issue in codes that are known to have none (MILC, UG4). Moreover, we were able to identify two scalability issues in a code that was thought to have only one (HOMME).

2.4 Case Study

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 [10] 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 literature mentions accurate models [7, 23] that describe the performance behavior of wavefront processes as they occur in Sweep3D on various architectures. The LogGP model reported in [7] characterizes the communication time as follows:

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

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:

$$t^{comm} = c \cdot \sqrt{p} \tag{3}$$

The (combined) model generated by our tool for the two receive operations involved in the wavefront process (sweep $\rightarrow$ MPI_Recv) is $3.99 \cdot \sqrt{p}$ and, thus, consistent with Equation (3). As Figure 2 illustrates, it also matches our measurements on Juqueen quite accurately.

In contrast to the growing execution time, the models for both the number of bytes and the number of messages received predict constant values independent of the number of processes. This 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.

2.5 Compiler-Driven Performance Modeling

In a similar but orthogonal subproject, we develop techniques for compiler-guided automated performance modeling. We use a mix of static analysis to count loop iterations and assess the theoretical scaling and parallelizability of practical codes [5] with dynamic multi-parameter performance model generation during runtime [1]. The static analysis instantiates work-depth models of parallel applications. It supports the large class of practically relevant loops with affine update functions and generates additional parameters for other expressions. The method can be used to determine whether the theoretically maximum parallelism is exposed in a practical implementation of a problem. The scheme over-approximates the performance of programs if loops are not affine or guards cannot be determined automatically. The dynamic approach under-approximates the program's behavior by analyzing particular executions. PEMOGEN, our compilation and modeling framework, automatically instruments applications to generate performance models during program execution. We used PEMOGEN to automatically detect 3,370 kernels from fifteen NAS and Mantevo applications and model their execution times. Both schemes were implemented in the Low Level Virtual Machine (LLVM) compiler framework [11].

This work is a first step towards full automation of the model generation. Open problems include non-linear combinations of different parameters as well as improved statistical techniques for model generation.

3 Workflow Manager

As illustrated in Figure 1, the identification of scalability bugs demands multiple executions of performance measurements, both with different and with identical

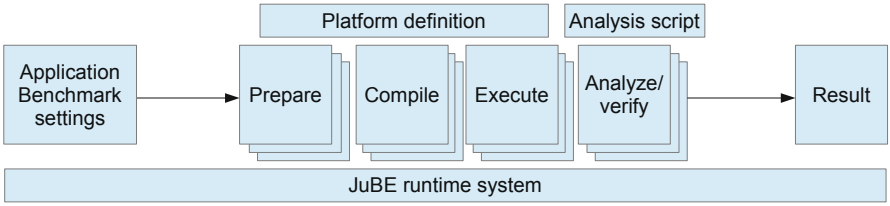

Fig. 3. JuBE workflow

input parameters, the latter to minimize the impact of jitter on shared machines. To automate this process, we use the Jülich Benchmarking Environment (JuBE) developed by Forschungszentrum Jülich. The steps carried out by JuBE are shown in Figure 3. The stacked boxes for preparation, compilation, execution, and analysis mean that these steps of the workflow might exist multiple times.

When running JuBE, it will perform the aforementioned steps in sequence. It is important to note that JuBE is able to easily create combinatorial runs of multiple parameters. For example, in a scaling experiment, one can simply specify multiple numbers of processes, and/or multiple threads per process, and JuBE will create one experiment for each possible combination, submit all of them to the resource manager, collect all results, and display them together.

4 Modeling OpenMP Performance

While scalability bugs are known issues for MPI applications and an MPI performance modeling methodology exists, it has not been applied to OpenMP and the interactions with MPI. As OpenMP represents the de-facto standard for exploiting manycore architectures, it will become of higher importance to exascale systems. Historically, multithreading and hence OpenMP usually did not require modeling, as it was easily possible to experimentally tests applications due to the limited amount of parallelism. With the ongoing trend of integrating more cores into CPUs, the level of parallelism rises and will most likely continue to rise well into the exascale era. Therefore, modeling OpenMP performance and detecting scalability bugs becomes important. Also understanding OpenMP modeling will enable to address hybrid applications, i.e., application using both MPI and OpenMP, which have to strike a careful balance distributing available compute resources between MPI processes and per-process OpenMP threads. Performance modeling could provide an answer to this question, indicating the sweet spot—without the need to experimentally test all possible thread and process combinations.

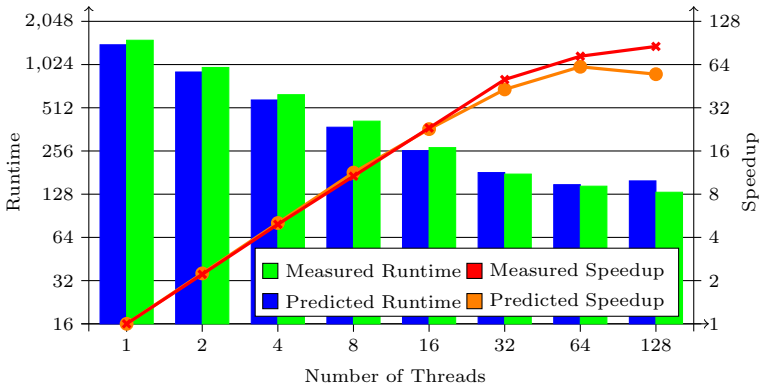

Fig. 4. OpenMP CG solver: comparison of measurements with the model

First OpenMP Modeling Experience

To determine possible model parameters and to ascertain precision, we analyzed a best effort implementation of a conjugate-gradient (CG) solver implementation from a recent OpenMP tuning study. In contrast to pure MPI applications, the regions required to sufficiently model the core are much smaller - typically comprising only the extent of single OpenMP constructs. On the other hand, the impact of resource limitations, such as memory bandwidth, and the impact of additional parallelism on the available resources is much more difficult to account for.

For our test code, we were able to manually create a fairly accurate model of the runtime using existing standard benchmarks. For this we measured memory bandwidth using the STREAM TRIAD benchmark and the runtime overhead of OpenMP constructs for each possible thread count on our test system, the BCS System of the RWTH Aachen University. We then combined these measurement results with an analytical model of the remaining computational parts to obtain the times shown in Figure 4 [19]. Our measurement results of an optimized kernel implementation were relatively close to the predicted runtimes, with some leeway owed to peculiarities of the STREAM memory benchmark. As this benchmark does not exhibit exactly the same memory access pattern as the CG solver, its measurements can only approximate the bandwidth used by the CG solver. As a result, especially for the first eight threads of the deployed eight-socket system and for the close to saturation levels at the peak capacity of the system (128 threads), the memory bandwidth available to each thread deviates more substantially, causing a higher deviation of the model from the predicted runtime. A better memory model would most certainly have reduced these effects. Overall, however, our experience shows that by combining per-thread measurements of memory bandwidth and OpenMP construct overhead with partial analytical modeling of the application a model of OpenMP performance can be constructed.

5 Scalability of the Multigrid Solver in UG4

A real-world application code for the performance analysis within the project is the UG4 software library. It is a general-purpose simulation framework for the grid-based solution of partial differential equations using finite element or finite volume methods and actively developed at the Goethe Center for Scientific Computing of University of Frankfurt. It is used to address a broad variety of problems arising in natural sciences such as biology, neuroscience, physics and engineering, including drug diffusion through human skin, signal transport in neurons, and several kinds of flow problems like Navier-Stokes flow or subsurface flow in porous media.

Because it is a relevant application from computational biology, we are focusing in this project on the drug transport through the human skin. The medicine diffusion is modeled in a 3d tetrakaidekahedra-based grid, resolving lipid bilayers and corneocyte cell components of the stratum corneum in detail [14, 15]. To get an idea, consider the simplified 2d brick-and-mortar model shown in Figure 5. While the transport is faster within the lipid bilayers, also the transport through the corneocytes is analyzed. To resolve the biological setting in detail, a 3d tetrakaidekahedral grid must be used. These delicate geometries need high resolutions and therefore require massively parallel computation.

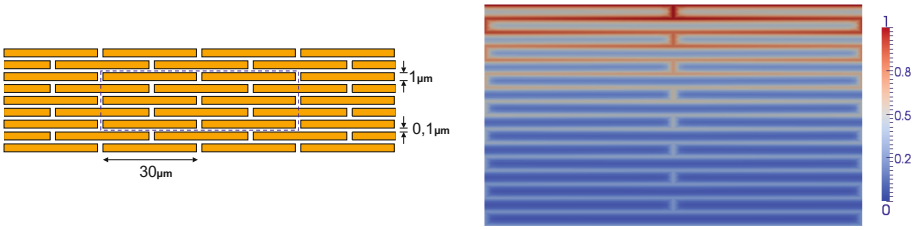

Fig. 5. Illustration of skin permeation. Left: The stratum corneum is build up by corneocyte cells (yellow) and lipid bilayers (channels). Right: medicine concentration diffusing from top to bottom.

Within a simulation, large sparse matrix systems arise that must be inverted. This part is not only one of the most time-consuming kernels of the application but also an algorithm that is hard to parallelize. We chose a geometric multigrid algorithm, since these are known to be of optimal complexity, i.e., its computational effort only increases linearly with the problem size, which makes it a promising candidate to achieve good weak-scaling results.

To generate performance models of the solver implementation, we performed weak-scaling runs for a diffusion problem on the Juqueen supercomputer using five identical runs for each process number to account for run-to-run variation. The analysis showed that no kernel in the application exhibited more than $O(log(p))$ growths in runtime. Hence, no scalability bugs were detected. This is

also in agreement with weak scaling studies performed on process configurations larger than the ones used to generate the models. This is a good starting point for a more complex and in-depth analysis in the future where we plan to analyze different matrix solver types and setups. In addition we want to analyze other metrics such as floating-point rates or message sizes, and apply the model generator to different physical settings.

6 Conclusion

In the Catwalk project, we have already made significant progress towards our original goal of automating key activities of the performance modeling process. Now, a lightweight tool exists that can be used to generate useful scalability models for arbitrarily complex MPI codes. Tests on a range of applications confirmed models reported in the literature in cases where such models existed, and also helped uncover a previously unknown scalability issue in another case.

In the future, we want to apply our approach to the co-design of exascale software and hardware. Co-designing applications with systems is a powerful technique to ensure early and sustained productivity as well as good system design. We want to assist this process by automating many of the back-of-the-envelope calculations involved in co-design with a lightweight requirements analysis for scalable parallel applications. We want to generate empirical models that allow projections not only for different numbers of processes but also for different problem sizes. System designers then can use the process-scaling models in tandem with the problem-scaling models and the specification of a candidate system to determine the resource usage of an application execution with a certain problem size.

Acknowledgment. The Catwalk project is funded by the DFG Priority Program 1648 *Software for Exascale Computing* (SPPEXA). Computing resources of the Jülich Supercomputing Centre and RWTH Aachen University are gratefully acknowledged.

References

1. Bhattacharyya, A., Hoefler, T.: PEMOGEN: Automatic Adaptive Performance Modeling during Program Runtime. In: To appear in Proc. of the 23rd Intl. Conference on Parallel Architectures and Compilation Techniques (PACT 2014). ACM (August 2014)
2. Boyd, E.L., Azeem, W., Lee, H.H., Shih, T.P., Hung, S.H., Davidson, E.S.: 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), pp. 188–192. IEEE Computer Society (1994), http://dx.doi.org/10.1109/ICPP.1994.30
3. Calotoiu, A., Hoefler, T., Poke, M., Wolf, F.: Using automated performance modeling to find scalability bugs in complex codes. In: Proc. of the ACM/IEEE Conference on Supercomputing (SC 2013). ACM, Denver (November 2013)

4. Carrington, L., Laurenzano, M., Tiwari, A.: Characterizing large-scale hpc applications through trace extrapolation. Parallel Processing Letters 23(4) (2013)
5. Hoefler, T., Kwasniewski, G.: Automatic Complexity Analysis of Explicitly Parallel Programs. In: To appear in Proc. of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2014). ACM (June 2014)
6. Hoefler, T., Gropp, W., Kramer, W., Snir, M.: Performance modeling for systematic performance tuning. In: State of the Practice Reports, SC 2011, pp. 6:1–6:12. ACM (2011), http://doi.acm.org/10.1145/2063348.2063356
7. Hoisie, A., Lubeck, O.M., Wasserman, H.J.: Performance analysis of wavefront algorithms on very-large scale distributed systems. In: Cooperman, G., Jessen, E., Michler, G. (eds.) Workshop on Wide Area Networks and High Performance Computing. LNCIS, vol. 249, pp. 171–187. Springer, Heidelberg (1999), http://dl.acm.org/citation.cfm?id=647259.720937
8. Ipek, E., de Supinski, B.R., Schulz, M., McKee, S.A.: An approach to performance prediction for parallel applications. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 196–205. Springer, Heidelberg (2005)
9. Kerbyson, D.J., Alme, H.J., Hoisie, A., Petrini, F., Wasserman, H.J., Gittings, M.: Predictive performance and scalability modeling of a large-scale application. In: Proc. of the ACM/IEEE Conference on Supercomputing (SC 2001), p. 37. ACM (2001)
10. Los Alamos National Laboratory: ASCI SWEEP3D v2.2b: Three-dimensional discrete ordinates neutron transport benchmark (1995), http://wwwc3.lanl.gov/pal/software/sweep3d/
11. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proc. of the Intl. Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization CGO 2004, p. 75. IEEE Computer Society, Washington, DC (2004), http://dl.acm.org/citation.cfm?id=977395.977673
12. Lee, B.C., Brooks, D.M., de Supinski, B.R., Schulz, M., Singh, K., McKee, S.A.: 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 2007), pp. 249–258. ACM (2007), http://doi.acm.org/10.1145/1229428.1229479
13. Mathis, M.M., Amato, N.M., Adams, M.L.: A general performance model for parallel sweeps on orthogonal grids for particle transport calculations. Tech. rep., College Station, TX, USA (2000)
14. Nägel, A., Heisig, M., Wittum, G.: The state of the art in computational modelling of skin permeation. Advanced Drug Delivery Systems (2012)
15. Nägel, A., Heisig, M., Wittum, G.: A comparison of two-and three-dimensional models for the simulation of the permeability of human stratum corneum. European Journal of Pharmaceutics and Biopharmaceutics 72(2), 332–338 (2009)
16. Petrini, F., Kerbyson, D.J., Pakin, S.: 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 2003), p. 55. ACM (2003), http://doi.acm.org/10.1145/1048935.1050204
17. Picard, R.R., Cook, R.D.: Cross-validation of regression models. Journal of the American Statistical Association 79(387), 575–583 (1984), http://www.tandfonline.com/doi/abs/10.1080/01621459.1984.10478083

18. Pllana, S., Brandic, I., Benkner, S.: 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), pp. 279–284 (2007)
19. Schmidl, D., Terboven, C., Iwainsky, C., Bischof, C., Müller, M.S.: Towards a performance engineering workflow for OpenMP 4.0. In: Performance Engineering MS bei ParCo 2013, Munich (2013)
20. Spafford, K., Vetter, J.S.: Aspen: a domain specific language for performance modeling. In: Proc. of the ACM/IEEE Conference on Supercomputing (SC 2012), p. 84 (2012), http://dl.acm.org/citation.cfm?id=2389110
21. Tallent, N.R., Hoisie, A.: Palm: easing the burden of analytical performance modeling. In: Proc. of the International Conference on Supercomputing (ICS), pp. 221–230 (2014), http://doi.acm.org/10.1145/2597652.2597683
22. Vetter, J.S., Worley, P.H.: Asserting performance expectations. In: Proc. of the ACM/IEEE Conference on Supercomputing (SC 2002), pp. 1–13 (2002), http://doi.acm.org/10.1145/762761.762809
23. Wasserman, H., Hoisie, A., Lubeck, O., Lubeck, O.: 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)
24. Wu, X., Mueller, F.: ScalaExtrap: trace-based communication extrapolation for SPMD programs. In: Proc. of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP 2011, pp. 113–122. ACM, New York (2011), http://doi.acm.org/10.1145/1941553.1941569
25. Lee, B.C., Brooks, D.M., de Supinski, B.R., Schulz, M., Singh, K., McKee, S.A.: 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 2007), pp. 249–258. ACM (2007), http://doi.acm.org/10.1145/1229428.1229479