Influence of Noisy Environments on Behavior of HPC Applications

D. A. Nikitenko,[1,*] F. Wolf,[2,] B. Mohr,[3,***] T. Hoefler,[4,****] K. S. Stefanov,[1,*****] Vad. V. Voevodin,[1,******] A. S. Antonov,[1,*******] and A. Calotoiu[4,********]**

(Submitted by E. E. Tyrtyshnikov)

[1]*Research Computing Center, Moscow State University, Moscow, 119991 Russia*
[2]*Laboratory for Parallel Programming, Technical University of Darmstadt, 64289 Darmstadt, Germany*
[3]*Juelich Supercomputing Centre, 52428 Juelich, Germany*
[4]*ETH Zurich, 8092 Zurich, Switzerland*

Received March 15, 2021

Abstract—Many contemporary HPC systems expose their jobs to substantial amounts of interference, leading to significant run-to-run variation. For example, application runtimes on Theta, a Cray XC40 system at Argonne National Laboratory, vary by up to 70%, caused by a mix of node-level and system-level effects, including network and file-system congestion in the presence of concurrently running jobs. This makes performance measurements generally irreproducible, heavily complicating performance analysis and modeling. On noisy systems, performance analysts usually have to repeat performance measurements several times and then apply statistics to capture trends. First, this is expensive and, second, extracting trends from a limited series of experiments is far from trivial, as the noise can follow quite irregular patterns. Attempts to learn from performance data how a program would perform under different execution configurations experience serious perturbation, resulting in models that reflect noise rather than intrinsic application behavior. On the other hand, although noise heavily influences execution time and energy consumption, it does not change the computational effort a program performs. Effort metrics that count how many operations a machine executes on behalf of a program, such as floating-point operations, the exchange of MPI messages, or file reads and writes, remain largely unaffected and—rare non-determinism set aside—reproducible. This paper addresses initial stage of an ExtraNoise project, which is aimed at revealing and tackling key questions of system noise influence on HPC applications.

2010 Mathematical Subject Classification: 68W10, 68M01, 68M20

Keywords and phrases: *high-performance computing, parallel computing, performance analysis, performance variability, supercomputers*

1. INTRODUCTION

High-performance computing is a key technology of the 21st century. Numerous application examples, ranging from the improved understanding of matter to the discovery of new materials and from the study of biological processes to artificial intelligence, give evidence of its tremendous potential. Mastery of this technology will decide not only on the economic competitiveness of a

* E-mail: dan@parallel.ru
** E-mail: felix.wolf@tu-darmstadt.de
*** E-mail: b.mohr@fz-juelich.de
**** E-mail: torsten.hoefler@inf.ethz.ch
***** E-mail: cstef@parallel.ru
****** E-mail: vadim@parallel.ru
******* E-mail: asa@parallel.ru
******** E-mail: alexandru.calotoiu@inf.ethz.ch

society but will ultimately influence everything that depends on it, including the society's welfare and stability. Moreover, there is broad consensus that high-performance computing is indispensable to address major global challenges of mankind such as climate change and energy consumption. However, the demand for computing power needed to solve problems of such enormous complexity is almost insatiable. In their effort to answer this demand, supercomputer vendors work alongside computing centers to find good compromises between technical requirements, tight procurement and energy budgets, and market forces that dictate the prices of key components. The results are sophisticated architectures that combine unprecedented numbers of processor cores into a single coherent system, leveraging commodity parts or at least their designs to lower the costs where in agreement with design objectives.

Exploiting the full power of HPC systems has always been hard and is becoming even harder as the complexity and size of systems and applications continues to grow. On the other hand, the savings potential in terms of energy and CPU hours that application optimization can achieve is enormous [1]. As the number of available cores increases at tremendous speed, reaping this potential is becoming an economic and scientific obligation. For example, a hypothetical exascale system with a power consumption of 20 MW (very optimistic estimate) and 5,000 h of operation per year would – assuming an energy price of EUR 0.1 per kWh – produce an energy bill of EUR 10M per year.

Ever-growing application complexity across all domains requires a continuous focus on performance to productively use the large-scale machines that are being procured. However, designing such large applications is a complex task demanding foresight since they require large time investments in development and verification and are therefore meant to be used for decades. Thus, it is important that the applications be efficient and potential bottlenecks are identified early in their design as well as throughout their whole life cycle. Continuous performance analysis starting in early stages of the development process is therefore an indispensable prerequisite to ensure early and sustained productivity.

In the past, powerful performance-analysis tools such as TAU [2], HPCToolkit [3], or Score-P/Scalasca [4][5] helped application developers achieve performance objectives. The typical workflow suggested by their designers is the following.

A user first instruments the code, and then executes it in the desired execution configuration, producing performance data usually in the form of profiles, which summarize execution time, possibly alongside other performance metrics, across the entire execution, or traces, which log performance-critical events with timestamps. Subsequently, interactive or fully automatic analyses pinpoint and explain performance problems, giving hints of how to remove them. Repeating the procedure with an optimized version of the code and comparing the results shows how effectively the code has been tuned. This methodology relies on the implicit assumption that shorter execution time (or less energy consumption) means better performance. This worked well, for example, on systems from the discontinued IBM Blue Gene series, which exposed their application to stable execution environments with only negligible degrees of interference, giving more or less consistent performance results across runs and allowing conclusions with statistical significance based on a few runs at most.

However, increasing performance variability challenges this classic tuning methodology. For example, Chunduri et al. report that execution times of some applications on Theta, a Xeon-Phi-based Cray XC system at Argonne National Laboratory, deviate from the minimum by up to 70% [6]. In general, performance variability, the difference between execution times across repeated runs of an application in the same execution environment [7], is the consequence of node-level effects, such as OS noise, dynamic frequency scaling, manufacturing variability, or shared-cache contention, and system-level effects, such as network and file-system congestion in the presence of concurrently running jobs. To some extent, the interference caused by contention for shared resources occurs by design because resource sharing is expected to improve system utilization. For example, Dragonfly networks [8] adaptively route traffic through less frequented partitions of the topology, potentially slowing down the communication of other jobs [9]. Power capping is another, actually desired feature that, as a side effect, introduces load imbalance in otherwise balanced computations [10], prolonging executions in unpredictable ways. Moreover, deepening storage hierarchies (e.g., with burst buffers as an intermediate layer) are on the one hand introduced to remove pressure from the backend file system, but on the other make I/O performance highly volatile. These examples demonstrate that performance variability is the price system designers pay to satisfy important optimization objectives and is therefore unlikely to disappear in the near future.

In addition to the immediate negative effects this variability has on performance, including load imbalance and jobs aborted because their execution time suddenly exceeded the scheduler-allowed maximum, variability also complicates performance analysis. Execution time can no longer be reported as a single number, but must rather be presented as a probability distribution. Drawing statistically significant conclusions requires many more experiments, substantially increasing both the cost and effort needed to motivate and validate tuning decisions. Because of the irregularity of interference from the perspective of the affected job, performance analysts often refuse to consider it an intrinsic part of application execution, which is why they wish to factor it out. Especially data-driven performance modeling [11] suffers as the resulting scaling models may reflect random noise and loose predictive power.

2. STATE OF THE ART

For many years now, performance interference—because of its perceived random nature often referred to as noise—has received considerable attention. In their seminal paper titled *The Case of the Missing Supercomputer Performance* [12], Petrini et al. describe how OS noise reduced application performance by a factor of two. Depending on their source, noise patterns vary in terms of intensity and frequency, ranging from continuous high-frequency background noise (e.g., OS noise) with low amplitude to more irregular forms with occasionally very high impact (e.g., file-system contention).

Strategies to reduce OS noise include core specialization [6] or improved system designs. For example, to lower OS noise, Cray and IBM introduced new kernel designs with significantly less interference potential, either by reducing daemons and improving memory allocation [13] or by creating a new lightweight kernel from bottom up [14]. On Dragonfly networks, optimized job placement can help mitigate the worst incidents of interference [9], while I/O-aware scheduling is seen as an instrument to tackle file-system congestion [15]. Nevertheless, as recent studies suggest [6, 7, 10, 16], performance variability is far from being eliminated, in spite of these attempts, and will remain an active area of research in the foreseeable future. In fact, Patki et al. even identified a tradeoff between performance optimality and reproducibility [7]. In addition to external interference, performance variability can also be caused by changes of the execution environment outside the control of the user such as different process-to-node mappings [17] or thread-to-core mappings on NUMA systems, which are per-se independent of noise, but may change the sensitivity of jobs to noise.

2.1. Performance measurement & analysis in noisy environments.

Porting, adapting and tuning applications to today's complex systems is a complicated and time-consuming task. Sophisticated integrated performance measurement, analysis, and optimization capabilities are required to efficiently utilize such systems.

Research on parallel performance tools has a long history. First tools appeared at the same time as the first parallel computer systems back in the 1980s and early 90s. Meanwhile, many performance instrumentation, measurement, analysis and visualization tools exist [18], e.g. TAU [2], HPCToolkit [3], Extrae/Paraver [19], Score-P [4], Vampir [20], and Scalasca [5]. These stand-out compared to other research tools as they are *portable*, *scalable*, *versatile* (i.e., they allow the performance analysis of all levels of today's HPC systems: message passing between nodes, multi-threading and multi-tasking inside nodes, and offloading to accelerators), and *supported* (i.e., there are well-established groups or organizations behind them which maintain and further develop them).

All measurement-based tools use the performance analysis workflow mentioned earlier. It relies on the fact that measurements can be repeated reliably; otherwise it is difficult, and in some cases impossible, to distinguish whether changes in the measurement data are the results of the tuning efforts of the developer or were produced by a noisy environment and system. The tools also lack information on the resource usage of concurrently running jobs that may possible interfere with the target program. Current best practice is noise avoidance or noise reduction, i.e., performing experiments in non-shared and controlled environments. If this is not possible, performance analysts have to perform a larger number of measurements, and then hope that the effects of noise in the measured data can be removed with the help of statistical methods. However, correctly designing insightful experiments to measure and report performance numbers is a challenging task. Hoefler and

Belli [21] summarize the best practices for scientific benchmarking of parallel computing systems, describe statistically sound analysis and reporting techniques, and present simple guidelines for experimental design in parallel computing.

There are only a few performance tools that try to help analyze noisy data. Typically they focus on network interference. The Ravel tool [22] uses a new trace visualization approach based on transforming the event history into logical time inferred directly from happened-before relationships, which better preserves and highlights event patterns and dependencies between processes. The original timing data is then encoded through color, leading to a more intuitive visualization. Zhou Tong et al. [23] present a novel trace-based analysis tool that rapidly classifies an MPI application as bandwidth-bound, latency-bound, load-imbalance-bound, or computation-bound for different interconnection networks. The tool uses an extension of Lamport's logical clock to track application process in the trace replay to infer the application characteristics of an application. Like the Ravel tool, it still uses real-time timestamps and other metrics to measure and model non-communication events, and therefore this part of the analysis is still sensitive to noise. In addition, both tools only work with MPI applications.

2.2. Performance modeling in noisy environments.

Classic performance analysis allows the programmer to observe application performance in actual runs, which can become expensive when exploring a larger configuration space, e.g., by varying core count and problem size, and impossible when the target platform is not yet available. One option to explore the performance on emerging architectures that exist only as blueprint is to use simulation tools [24], however, depending on the level of detail, building simulators and running simulations also can be quite expensive. Performance models, in contrast, allow the configuration space of an application to be explored much faster, while also giving access to unavailable execution configurations and theoretical platform designs. A performance model is a formula that expresses a performance metric of interest such as execution time or energy consumption as a function of one or more execution parameters such as the size of the input problem or the number of processors. Note that the popular roofline model [25], which visualizes a hardware-constrained performance ceilings for varying values of operational intensity, is just a very specialized instantiation of this definition. Although often based on simplifying assumptions, performance models offer valuable insight at the small cost of evaluating an arithmetic expression. Hoefler et al. describe a six-step process to guide the (manual) creation of analytical performance models [26]. However, deriving such models analytically from the code is still so laborious that too many application developers shy away from the effort.

To ease the burden, performance modeling techniques with varying degrees of automation have been introduced [27–29]. Many state-of-the-art tools support empirical performance modeling, a method which derives performance models from measurements [30–32], an approach that is also popular in application areas outside HPC such as databases [33] or software product lines [34]. To generate performance models from measurements, a range of techniques is applied [35], many of them classifiable as machine learning, including regression, artificial neural networks, and other statistical methods.

The primary way of addressing noise is the same as for performance measurement in general: repeating measurements and trying to find a representative value such as the minimum or median. Another successful approach that comes on top is the introduction of a prior into the learning process, limiting the set of discoverable functions, for example, to polynomials and/or logarithms [11, 30, 31]. However, our experience suggests that this is not enough when run-to-run variation is high — especially if the performance model has multiple parameters [36]. Recently, Duplyakin et al. [37] applied Gaussian process regression (GPR) [38] to the field of performance modeling. Here, selecting a higher absolute value for the noise value hyper-parameter can help GPR to better cope with noisy data, reducing the risk of overfitting, but at the cost of generating "smoother" models that do not necessarily capture all behaviors present in the data.

2.3. *Characterization of noise & understanding the noise sensitivity of applications.*

Studies of noise on HPC systems and its influence on parallel program execution [39–42] consider different sources of noise independently, but do not take into account the possible synchronization of noise events across the system. Afzal et al. [40] analyze how a single noise event spreads across the system, but do not extend their work to several or periodic events. However, some events in a computing system may occur simultaneously or almost simultaneously. In addition to events that are intentionally synchronized such as running checks on multiple nodes of the system, there is also a known tendency for independent events to become synchronized [43]. But if the events that cause delays in program execution are not independent, the overall influence on performance will likely differ and models that do not take into account possible noise synchronization may produce incorrect results. For example, the NWPerf monitoring system [44] synchronizes the moments when agents trigger actions on computational nodes, which, according to the authors, reduces the influence of these actions on the workload. But it is still not entirely clear how the influence of noise synchronization can be quantified.

One way of studying the impact of noise is injecting artificial noise [45]. To the best of our knowledge, however, present noise injections systems do not yet support synchronized artificial noise.

2.4. *Understanding the sensitivity of algorithms to noise.*

Because algorithms are the center piece of computer programs, primarily responsible for their resource consumption, catalogs have emerged in which researchers document the structure and characteristics of computational algorithms and their implementations for various computing platforms, including Templates for the Solution of Linear Systems[1] [39], ALGLIB[2], and A Library of Parallel Algorithms[3]. A distinctive feature of AlgoWiki[4] is the use of a single structure for the description of any algorithms, a concept of "problem-method-algorithm-implementation" chains [46], and special emphasis on the properties associated with parallelism.

The dynamic properties of algorithm implementations, in particular those related to scalability, are an important part of many scientific studies. Since noise is known to disturb load balance, it also presents a serious impediment to scalability [39]. There are a number of metrics for measuring scalability of algorithms, such as iso-efficiency [47]. Except for rare studies with a narrower focus such as tasking on shared-memory systems [48], these metrics characterize algorithms on a rather theoretical level and largely ignore the dynamic characteristics of their implementation on specific computing systems. And no universally accepted metric has yet been proposed that allows the scalability of algorithms implementations to be compared based on their dynamic characteristics. AlgoWiki advocates a multidimensional scalability metric [49], which was partially used to compare implementations of the algorithms described in AlgoWiki w.r.t. scalability. However, further studies revealed several disadvantages of this metric, mainly its significant dependence on the choice of the range of parameter values. Some modifications of this metric were proposed [50], which, however, did not solve all problems.

Most important for this project, all well-known algorithm catalogs describe the properties of numerical algorithms and their implementations without taking their response to noise into account, which can change their dynamic properties quite significantly. Considering the influence of environmental noise on program execution can draw a more realistic picture and open add a new dimension to the filed of algorithm engineering.

[1] http://www.netlib.org/linalg/html_templates/Templates.html
[2] https://www.alglib.net/
[3] https://www.cs.cmu.edu/~scandal/nesl/algorithms.html
[4] https://algowiki-project.org/en/

3. THE EXTRANOISE PROJECT

The recently started ExtraNoise joint project addresses the questions mentioned above. It is funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) and the Russian Foundation for Basic Research (RFBR). In addition to making performance analysis more noise resilient, the partners also aim at a better understanding of how applications respond to noise in general and which design choices increase or lower their active and passive interference potential. The key contributors of the project are TU Darmstadt, Juelich Supercomputing Centre, and Moscow State University. ETH Zurich also provides valuable expertise.

The overarching goal of the ExtraNoise project is making application performance analysis on systems with high performance variation both cheaper and more reliable—noise resilient in one word. The project targets a range of typical performance analysis techniques, including raw performance measurement, trace analysis, and empirical performance modeling. Expecting that our approach to noise resilience is to some degree system and application specific, we also want to better understand both noise patterns applications are exposed to on a given system and the assess the noise sensitivity of applications. From knowledge of noise sensitivity we want not only to derive strategies of how to conduct performance analysis most effectively but also give feedback for application developers on how to lower the impact of noise without compromising algorithmic performance.

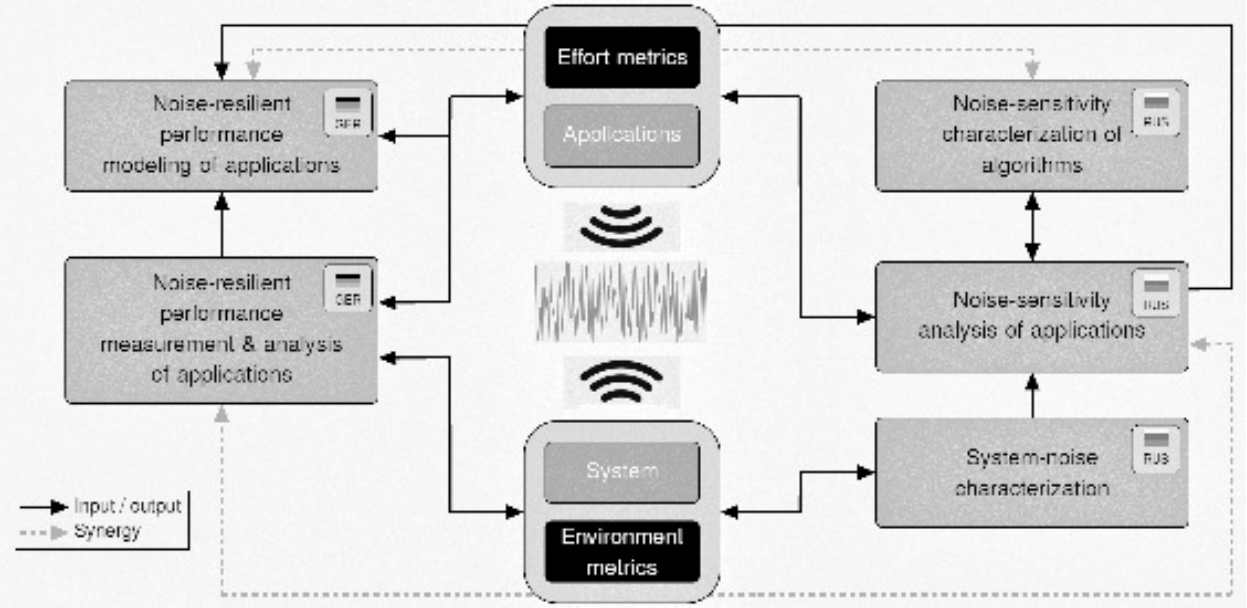

Figure 1. Performance analysis of HPC applications in noisy environments. The flag symbols on the work-package boxes indicate the country of the lead team.

Figure 1 illustrates our project concept. The grey boxes represent the five specific contributions we make:

Noise-resilient performance measurement & analysis of applications. Capture essential traits of the performance behavior in a noise-resilient way and, based on these traits, identify and quantify classic performance bottlenecks such as load imbalance (WP1).

Noise-resilient performance modeling of applications. Use noise-resilient performance measurements as prior to help empirical performance modeling ignore noise effects and investigate to which extent asymptotic scaling behavior can be derived without reliance on noise-affected performance metrics altogether (WP2).

System noise characterization. Measure and characterize the noise patterns an application is exposed to at runtime with a focus on I/O and interconnect noise sources (WP3).

Noise-sensitivity analysis of applications. Finds ways to determine how noise-sensitive an application is and why and use this knowledge for further performance analysis and optimization (WP4).

Noise-sensitivity characterization of algorithms. Distill the lessons learned about the noise sensitivity of applications into general insights into the noise sensitivity of algorithms and their implementation alternatives (WP5).

In the concept figure, black arrows represent input-output relationships. Light arrows indicate synergy potential in terms of common base concepts. As you can notice, each of the contribution is a focus of a dedicated work package.

To reach our goals, we build upon and extend established tools developed by partner organizations, namely JobDigest, Score-P, Scalasca, and Extra-P, which are described under preliminary work. The resulting methods will be evaluated with a combination of synthetic data, benchmarks, and modern production codes from both countries.

Below, let us define work packages that explain how we address the different research questions, not saying about coordination and dissemination.

4. REVEALING THE NOISE INFLUENCE ON APPLICATIONS WITH SYSTEM MONITORING

4.1. Measuring and Characterizing the Noise

The objective of this work package is to devise techniques for characterizing noise in HPC system. We focus on noise that originates from several nodes simultaneously, network activity or I/O. Special attention will be paid to noise that is synchronized across multiple nodes of a cluster.

As a first task, we will develop methods for detecting such noise. We will design suitable probes that can capture a variety of noise patterns. Probes will exploit synchronizing operations such as MPI collectives to expose noise-induced delays. The probes will be validated with artificial noise injected into the system, for example, using Gremlins [45]. Beyond existing noise-injection methods, we will also create a testbed with synchronized artificial noise.

The second task is to study how the noise level changes over time. The noise may change not only because of interference from other jobs, but also as a result of housekeeping procedures. We start with considering these latter changes.

The third task is to develop environmental metrics which can show the system noise level without the need to run dedicated tests. This will then also expose the noise originating from other jobs interfering with the one of interest. These metrics will be integrated into the reports issued by our monitoring system JobDigest. Thus, we will make it possible to evaluate the noise level on the system (counters representing various noise sources like interconnect, I/O, etc.) during an application run and help estimate the resulting performance degradation without instrumenting the code and making restricting assumption about the application structure like in our earlier work [51].

This work package is based on system monitoring data and data collected during test runs. The output can be used to develop methods to analyze the noise sensitivity of applications and to estimate the impact of noise on their runtime.

4.2. Noise-Sensitivity Analysis of Applications

The objective of this work package is to develop methods for detecting and understanding noise-related application behavior — both as a target and a producer of noise — based on the analysis of system monitoring data. The central idea is to learn how the change of environment metrics over time indicates noise-induced changes of the application behavior. To this end, we plan to solve three tasks.

As the first task and main theme, we will correlate the evolution of environment metrics over time with changes in application performance. JobDigest will collect the necessary environment metrics using its underlying system-monitoring infrastructure, based on the results of the work package described in subsection 4.1. This will allow us to identify the most sensitive and the most disturbing applications on a given system and also quantify their active and passive interference potential.

The goal of the second task is to transfer the insights we gain from the analysis of specific applications in the first task to previously unseen applications. Specifically, we will employ machine learning to identify applications with similar noise-response behavior or similar active interference histories. This is supposed to broaden and accelerate the identification of applications with high degrees of noise sensitivity and/or potential to disturb others. We want to exploit such similarities (i) to determine how to modify applications to reduce both their active and passive interference potential and also (ii) to predict the noise-related behavior of applications at least to some degree, for example, as a hint to the scheduler as to which jobs should be run together and which not.

Based on the results of the previous two tasks, we plan to eventually integrate functionality into JobDigest to support application noise-sensitivity analysis. The integrated solution will be made available to supercomputer users, allowing us to collect their feedback on how to further tune and upgrade JobDigest.

This work package will use mechanisms developed and data produced in the work package described in subsection 4.1, including environment metrics and methods for measuring noise. At the same time, it will contribute to the work package described in subsection 4.3, providing means to assess the noise-sensitivity of algorithm implementations.

4.3. Noise-Sensitivity Characterization of Algorithms

The objective of this part is to characterize the noise sensitivity of algorithms and their implementations based on observations and theoretical estimations. The results will be added to the AlgoWiki Open Encyclopedia of Parallel Algorithmic Features in the form of extended algorithm descriptions to support scalable algorithm engineering and software development.

To characterize noise-related bottlenecks of canonical algorithm implementations, we are going to develop new schemata to describe such dynamic properties in a uniform way, on the basis of observations. Particular emphasis should be placed on how scaling properties respond to noise. We will evaluate how noise-resilient performance models can support such descriptions.

After that, we are going to study and describe the noise-related properties of implementations of various algorithms from the AlgoWiki encyclopedia. As our methods and tools for identifying and assessing the effect of noise on software mature, we plan to further expand the descriptions of noise sensitivity. Algorithm implementations with varying degrees of resistance to external noise will be highlighted. We will ponder the question whether resistance to external noise is a characteristic of only specific implementations or whether it can be attributed to more general algorithmic features, to methods for solving problem classes, or even to the problems themselves.

The proposed work will rely on the experimental noise-sensitivity analysis of various algorithm implementations. While noise-resilient performance models may contribute to more comprehensive descriptions of algorithm implementations, insights into the noise resistance of algorithm implementations may support the performance-modeling tool chain in selecting a balanced prior that is neither too rigid nor too flexible. Moreover, such insights can point application performance analysts to implementation details that would be missed otherwise.

5. CONCLUSIONS

The ExtraNoise project, recently supported by DFG and RFBR, aims vital questions of performance evaluation in real-life noisy environments. The international team has got a successful experience in collaborating with each other, and possesses a rich project-related background and expertise. The three years of project promise to provide valuable output. At present, at a kickoff point, we encourage the reader to share interesting use cases of noise influence and to collaborate.

Acknowledgments. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University [52]. The reported study was funded by RFBR, project number 21-51-150001, and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 449683531.

REFERENCES

1. C. Bischof, D. An Mey, and C. Iwainsky. Brainware for green HPC. *Computer Science-Research and Development*, 27(4):227–233, 2012.
2. S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.
3. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrant Computation: Practice & Experience*, 22(6):685–701, April 2010.
4. D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Yu. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, pages 85–97. Gauß-Allianz, Springer, 2012. DOI 10.1007/978-3-642-24025-6_8.
5. 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. DOI 10.1002/cpe.1556.
6. S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran. Run-to-run variability on Xeon Phi based Cray XC systems. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC17), Denver, CO, USA*, SC вЂ™17, New York, NY, USA, 2017. Association for Computing Machinery.
7. T. Patki, J. J. Thiagarajan, A. Ayala, and T. Z. Islam. Performance optimality or reproducibility: That is the question. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'19, New York, NY, USA, 2019. Association for Computing Machinery.
8. J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *Proc. of the 35th International Symposium on Computer Architecture (ISCA)*, pages 77–88. IEEE, 2008.
9. X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan. Watch out for the bully! job interference study on dragonfly network. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC16), Salt Lake City, UT, USA*, pages 750–760. IEEE, November 2016.
10. Yu. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Ya. Wada, K. Fukazawa, M. Ueda, M. Kondo, and I. Miyoshi. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC15), Austin, TX, USA*, pages 1–12. ACM, November 2015.
11. 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 International Conference for High Performance Computing, Networking, Storage and Analysis (SC13), Denver, CO, USA*, pages 1–12. ACM, November 2013. DOI 10.1145/2503210.2503277.
12. 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 Supercomputing Conference (SC03), Phoenix, AZ, USA*, November 2003.
13. D. Wallace. Compute node linux: Overview, progress to date & roadmap. In *Proc. of the 49th Cray User Group Conference, Seattle, WA, USA*, May 2007. `https://cug.org/5-publications/proceedings_attendee_lists/2007CD/S07_Proceedings/pages/Authors/Wallace-12B/Wallace-12B_paper.pdf`.
14. M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from blue gene's cnk. In *Proc. of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10), New Orleans, LA, USA*, pages 1–10, Nov 2010.
15. S. Herbein, D. H. Ahn, D. Lipari, T. R.W. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer. Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters. In *Proc. of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, HPDC'16, page 69–80, New York, NY, USA, 2016. Association for Computing Machinery.
16. A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci. Taming performance variability. In *Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Carlsbad, CA, USA*, pages 409–425, October 2018.
17. A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*, pages 1–12. ACM, November 2013.
18. B. Mohr. Scalable parallel performance measurement and analysis tools — state-of-the-art and future challenges. *Supercomputing frontiers and innovations*, 1(2):108–123, 2014.

19. H. Servat, G. Llort, J. Giménez, and J. Labarta. Detailed performance analysis using coarse grain sampling. In Hai-Xiang Lin, Michael Alexander, Martti Forsell, Andreas Knüpfer, Radu Prodan, Leonel Sousa, and Achim Streit, editors, *Euro-Par 2009 — Parallel Processing Workshops*, pages 185–198, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
20. A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
21. T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15, New York, NY, USA, 2015. Association for Computing Machinery.
22. K. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann. Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics*, 20:2349–2358, 12 2014.
23. Z. Tong, S. Pakin, M. Lang, and X. Yuan. Fast classification of mpi applications using lamport's logical clocks. *Journal of Parallel and Distributed Computing*, 120, 05 2018.
24. C. Gómez, F. Martinez, A. Armejach, M. Moretó, F. Mantovani, and M. Casas. Design space exploration of next-generation hpc machines. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil*, pages 54–65. IEEE, 2019.
25. S. Williams, A. Waterman, and D. Patterson. An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4), 2009.
26. T. Hoefler, W. Gropp, W. Kramer, and M. Snir. Performance modeling for systematic performance tuning. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, pages 6:1–6:12. ACM, 2011.
27. N. R. Tallent and A. Hoisie. Palm: Easing the burden of analytical performance modeling. In *Proc. of the 28th ACM International Conference on Supercomputing*, ICS'14, pages 221–230, New York, NY, USA, 2014. ACM.
28. K. L. Spafford and J. S. Vetter. Aspen: A domain specific language for performance modeling. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC'12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
29. J. Hammer, G. Hager, J. Eitzinger, and G. Wellein. Automatic loop kernel analysis and performance modeling with kerncraft. *CoRR*, abs/1509.03778, 2015.
30. S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Proc. of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE)*, pages 395–404, New York, NY, USA, 2007. ACM.
31. L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, February 2006.
32. A. Grebhahn, C. Rodrigo, N. Siegmund, F. J. Gaspar, and S. Apel. Performance-influence models of multigrid methods: A case study on triangular grids. *Concurrency and Computation: Practice and Experience*, 29(17):e4057, 2017.
33. B. Hilprecht, C. Binnig, T. Bang, M. El-Hindi, B. HΓ¤ttasch, A. Khanna, R. Rehrmann, U. RΓ¶hm, A. Schmidt, L. Thostrup, and T. Ziegler. Dbms fitting: Why should we learn what we already know? In *Proc. of the 10th Conference on Innovative Data Systems Research (CIDR), Amsterdam, Netherlands*, January 2020.
34. N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3):487–517, Sep 2012.
35. B. C. Lee, D. M. Brooks, B. R De Supinski, M. H. Schulz, K. Singh, and S. 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), San Jose, CA, USA*, pages 249–258, March 2007.
36. M. Ritter, A. Calotoiu, S. Rinke, T. Reimann, T. Hoefler, and F. Wolf. Learning cost-effective sampling strategies for empirical performance modeling. In *Proc. of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA*. IEEE Computer Society, May 2020. (to appear).
37. D. Duplyakin, J. Brown, and R. Ricci. Active learning in performance analysis. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 182–191, September 2016.
38. A. McHutchon and C. E. Rasmussen. Gaussian process training with input noise. In *Proc. of the 24th International Conference on Neural Information Processing Systems (NIPS), Granada, Spain*, pages 1341вЂ“1349, Red Hook, NY, USA, 2011. Curran Associates Inc.

39. 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 International Conference for High Performance Computing, Networking, Storage and Analysis (SC10), New Orleans, LA, USA*, pages 1–11, November 2010.
40. A. Afzal, G. Hager, and G. Wellein. Propagation and decay of injected one-off delays on clusters: A case study. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, September 2019.
41. S. Agarwal, R. Garg, and N. K. Vishnoi. The impact of noise on the scaling of collectives: A theoretical approach. In D. A. Bader, M. Parashar, V. Sridhar, and V. K. Prasanna, editors, *High Performance Computing – HiPC 2005*, pages 280–289, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
42. P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, mar 2008.
43. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking*, 2(2):122–136, apr 1994.
44. R. Mooney, K.P. Schmidt, and R.S. Studham. NWPerf: a system wide performance monitoring tool for large Linux clusters. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pages 379–389. IEEE, 2004.
45. M. Schulz, J. Belak, A. Bhatele, P.-T. Bremer, G. Bronevetsky, M. Casas, T. Gamblin, K. E. Isaacs, I. Laguna, J. Levine, V. Pascucci, D. Richards, and B. Rountree. *Parallel Computing: Accelerating Computational Science and Engineering*, chapter Performance analysis techniques for the exascale co-design process, pages 19–32. Advances in Parallel Computing. IOS Press, 2014.
46. A. Antonov, A. Frolov, I. Konshin, and V.Voevodin. Hierarchical domain representation in the algowiki encyclopedia: From problems to implementations. In *Parallel Computational Technologies*, volume 910 of *Communications in Computer and Information Science*, pages 3–15. SPRINGER, 2018.
47. A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.
48. S. Shudler, A. Calotoiu, T. Hoefler, and F. Wolf. Isoefficiency in practice: Configuring and understanding the performance of task-based applications. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Austin, TX, USA*, pages 131–143. ACM, February 2017.
49. A. Antonov and A. Teplov. Generalized approach to scalability analysis of parallel applications. *Lecture Notes in Computer Science*, 10049:291–304, 2016.
50. P. Valkov, K. Kazmina, and A. Antonov. Using empirical data for scalability analysis of parallel applications. In *Communications in Computer and Information Science*, volume 1063, pages 58–73. Springer International Publishing, 2019.
51. A. Shah, M. S. Müller, and F. Wolf. Estimating the impact of external interference on application performance. In *Proc. of the 24th Euro-Par Conference, Turin, Italy*, volume 11014 of *Lecture Notes in Computer Science*, pages 46–58. Springer, August 2018.
52. V. Voevodin, A. Antonov, D. Nikitenko, P. Shvets, S. Sobolev, I. Sidorov, K. Stefanov, V. Voevodin and S. Zhumatiy, "Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community", *Supercomputing Frontiers and Innovations*, **6** 2, 4–11 (2019). DOI: 10.14529/jsfi190201