Engineering Algorithms for Scalability through Continuous Validation of Performance Expectations

Sergei Shudler, Yannick Berens, Alexandru Calotoiu, Torsten Hoefler, *Member, IEEE*, Alexandre Strube, and Felix Wolf

Abstract—Many libraries in the HPC field use sophisticated algorithms with clear theoretical scalability expectations. However, hardware constraints or programming bugs may sometimes render these expectations inaccurate or even plainly wrong. While algorithm and performance engineers have already been advocating the systematic combination of analytical performance models with practical measurements for a very long time, we go one step further and show how this comparison can become part of automated testing procedures. The most important applications of our method include initial validation, regression testing, and benchmarking to compare implementation and platform alternatives. Advancing the concept of performance assertions, we verify asymptotic scaling trends rather than precise analytical expressions, relieving the developer from the burden of having to specify and maintain very fine-grained and potentially non-portable expectations. In this way, scalability validation can be continuously applied throughout the whole development cycle with very little effort. Using MPI and parallel sorting algorithms as examples, we show how our method can help uncover non-obvious limitations of both libraries and underlying platforms.

Index Terms—Software engineering, high performance computing, parallel programming, performance analysis, performance modeling

1 Introduction

The most powerful supercomputers today allow computations to be run on tens of millions of cores and in the not-so-distant future this number may even grow to billions of cores [1]. Since many applications critically depend on parallel libraries, such as MPI, PETSc, ScaLAPACK, or HDF5, the scalability of these libraries is of utmost importance for reaching performance targets at scale. This becomes even clearer considering that application developers may be able to remove performance bottlenecks from their own code, but may find it more challenging to remove these bottlenecks from the libraries they are using.

Library developers, on the other hand, are confronted with the problem of continuous scalability validation as their code base evolves. In the past, they often did this by scaling the library to the full extent of the largest machine available to them, after which they manually compared the results with theoretical expectations. This is expensive in terms of both machine time and manpower. In cases where the library encapsulates complex algorithms that are the product of years of research, such expectations often exist in the form of analytical performance models [2], [3], [4]. However, translating such abstract models into concrete verifiable expressions is hard because it requires knowing all constants and restricts function domains to performance metrics that are effectively measurable on the target system. If only the asymptotic complexity is known, as is very commonly the case, this is in fact impossible. And if such a verifiable expression exists, it must be adapted every time the test platform is replaced and performance metrics and constants change.

To mitigate this situation, we combine empirical performance modeling with performance expectations in a novel scalability test framework. As depicted in Fig. 1, the framework adds empirical performance modeling to the test phase in the software development cycle, thereby introducing a new software engineering approach. Similar to performance assertions [5], our framework supports the user in the specification and validation of performance expectations. However, rather than formulating precise analytical expressions involving measurable metrics, the user has to only provide the asymptotic growth rate of the function/metric pair in question, making this a simple but effective solution for future exascale library development. We generate performance models similar to Calotoiu et al. [6]. However, instead of creating scaling models independently from the expected behavior as they do, we tailor the model search spaces to expectations and also generate divergence models that help

- *S. Shudler is with Argonne National Laboratory, Lemont, IL 60439. E-mail: sshudler@anl.gov.*
- *Y. Berens, A. Calotoiu, and F. Wolf are with Technische Universität Darmstadt, Darmstadt 64289, Germany. E-mail: yannick.berens@stud.tu-darmstadt.de, {calotoiu, wolf}@cs.tu-darmstadt.de.*
- *T. Hoefler is with ETH Zürich, Zürich 8092, Switzerland. E-mail: htor@inf.ethz.ch.*
- *A. Strube is with Jülich Supercomputing Centre, Jülich 52425, Germany. E-mail: a.strube@fz-juelich.de.*

Manuscript received 15 June 2018; revised 9 Jan. 2019; accepted 18 Jan. 2019. Date of publication 1 Feb. 2019; date of current version 8 July 2019.
(Corresponding author: Sergei Shudler.)
Recommended for acceptance by M. Yasugi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2019.2896993

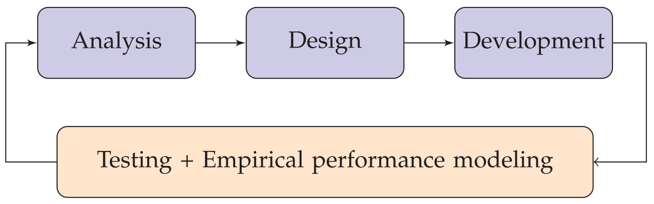

Fig. 1. Software development cycle with empirical performance modeling.

in understanding how the difference between expected and actual behavior would evolve as the number of processes increases. Moreover, in the absence of a clear expectation, the framework is able to supply the status quo as a substitute. This is especially useful during regression testing when the main task is to prevent new modifications from reducing scalability. A performance model generator combined with an automated workflow manager makes sure that the actual and expected behavior can be continuously compared.

The framework we propose can also be used in algorithm engineering [7]. Traditionally, algorithm theory does not focus on implementation and leaves this part to application development. However, growing complexities of both the algorithms and the hardware (e.g., parallelism, memory hierarchy, etc.) create a gap between promising algorithmic ideas and their practical use. Algorithm engineering, therefore, aims to bridge this gap by adopting elements from software engineering. In other words, it defines a cycle that consists of four major phases, namely, design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses. We show that our methodology can be beneficial to the algorithm engineering cycle in a number of ways.

Use cases of our framework include initial validation, regression testing, and benchmarking to compare implementation and platform alternatives. Although our work is not restricted to a specific type of software, we focus on library development because of its high impact and the greater availability of theoretical performance models. In comparison to the state of the art, we make the following specific contributions:

- An approach to engineer libraries and algorithms for extreme-scale systems based on incorporating empirical performance modeling into the development process
- Continuous scalability validation based on simple asymptotic growth rates, which are often easier to obtain than fully evaluable analytical expressions
- Generation of divergence models to characterize deviation as a function of the number of processes
- Targeted model search through expectation-driven construction of the search space
- Automatic workflow including execution of performance experiments and generation of performance models
- Testing whether the scaling behavior of the library is consistent across different functions

In the first case study, involving several MPI implementations, we demonstrate how our framework can be applied to (i) uncover growing memory consumption, (ii) reveal architectural constraints that limit the performance of a wide range of collective operations, and (iii) predict the violation of MPI performance guidelines. In the case study involving the Merging of Adaptive Finite IntervAls (MAFIA) code [8], we demonstrate that our approach is also applicable to algorithmic modeling. In this case, the model is a function of an algorithm parameter. Furthermore, we use the framework to validate the performance of parallel sorting algorithms.

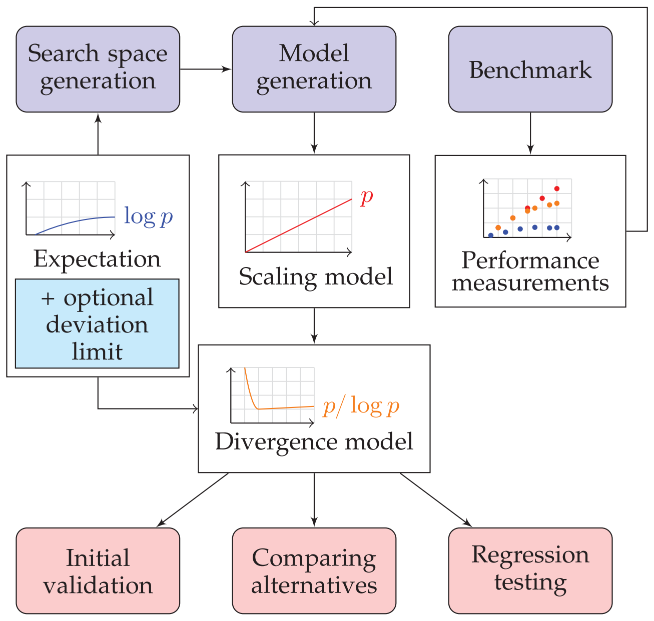

Fig. 2. Framework overview including use cases.

This work extends a previously published paper [9]. It adds the evaluation of Intel MPI and Open MPI to the main case study, as well as introduces a new case study of parallel sorting algorithms. The evaluation is performed on two systems, including a Blue Gene/Q machine, and highlights the applicability of our methodology to algorithm engineering. This paper also includes an extended discussion of the scalability validation workflow, the benchmark design in the MPI case study, and related work.

2 Scalability Validation Framework

The objective of our approach, which is illustrated in Fig. 2, is to provide insights into the scaling behavior of a library with as little effort as possible. It includes the following four steps: (i) *define expectations*; (ii) *design benchmark*; (iii) *generate scaling models*; and (iv) *validate expectations*. The first two are manual because they involve user decisions, while the second two are automatic. We describe each of them in detail below.

2.1 Define Expectations

We aim to keep our method simple and effective: it has to be usable in various settings with only an approximate idea of the expected result. For example, it is very unlikely that a programmer of a matrix-matrix multiplication can tell the floating-point rate or the achieved memory bandwidth for a given matrix size n. Thus, these metrics may be less useful in practice. However, every programmer will know whether he used the simple $\mathcal{O}(n^3)$ algorithm or Strassen's $\mathcal{O}(n^{2.8074})$ algorithm. Therefore, we let the user define expectations in big O notation (aka Landau notation). For some functions, one could even formulate a hypothetical (black-box) expectation, that is, an expectation based on either an approximation or an incomplete knowledge of the algorithm. For example, without any additional information about a library call `sort(int *array, int n)` for sorting an integer array, one might formulate a hypothetical expectation of $\mathcal{O}(n \log n)$, although the actual algorithm might require $\mathcal{O}(n^2)$ steps in the worst case.

In our expectation-centric performance modeling approach, the user does not have to be a domain expert to provide expectations. An initial guess of the scalability or a hypothetical expectation is enough for the scalability validation framework. However, before being able to define expectations, the user has to choose the library functions that will be subjected to the scaling analysis and the relevant scaling metrics. The more functions the user selects the more expensive it will become to construct the benchmark, which is why it can make sense to restrict the selection to those deemed most relevant. On the other hand, making too narrow a choice poses the risk of overlooking hidden scalability issues. Another important decision concerns the selection of scaling metrics. For some rarely called functions, memory consumption might be the primary concern, but for many others it will probably be runtime or floating-point operations. In general, we can distinguish between measured metrics such as runtime and metrics that can be counted as discrete units such as floating-point operations. Very often, the latter yield better empirical scaling models because they are less prone to jitter. If only a hypothetical expectation is available, as in the sorting example above, the model generator can use it to generate a model that better describes the current behavior. This model can then become the new expectation. This is especially useful when the user has little knowledge of the library or during regression testing when the main task is to prevent later modifications from introducing scalability bugs.

Sometimes, the functionality offered by one library function is a subset or a superset of the functionality offered by another library function. Or a library API may offer convenience functions with functionality that can be regarded as a short cut for a combination of other API functions. In such cases, it is possible to define optional *cross-function rules* that specify relationships between the scaling behavior of different functions [10]. For example, a short cut should not scale worse than the spelled-out implementation.

2.2 Design Benchmark

The benchmark must provide or generate valid library inputs and measure the selected performance metrics for the selected functions in various execution configurations (e.g., different numbers of processes or input sizes).

Occasionally, unexpected architectural constraints such as the network topology may increase the observable complexity of an implementation—without such factors, the software could be blamed in the sense of a performance bug that requires a fix. To help distinguish such effects from programming bugs, it is advisable to manually re-implement one or more representative library functions in a way that has been proven to show the expected behavior under ideal conditions—for example, using a known optimal algorithm from the literature. The difference between this *performance litmus test* and the original library functions is that the tester can usually trust the replica more than the original function because he thoroughly knows its internals. Should the original library function now show performance deviations, they can be compared with the results obtained for the litmus test. A similar deviation observed for the replica could then be seen as a strong indicator of architectural constraints that might also influence the behavior of other regular library functions. We discuss an example as part of our first case study in Section 3.1.

2.3 Generate Scaling Models

Our expectation-centric performance modeling approach assumes that the user provides an initial expectation function $E(x)$. Together with this expectation the user either provides a deviation limit $D(x)$ or a default deviation is chosen automatically. Looking at how most computer algorithms are designed and their complexity, we can identify a number of function classes with distinct rates of growth.

$$F_1(x) = \left\{\log_2^{i_1} x\right\}$$
$$F_2(x) = \left\{\, x^{i_2}\right\}$$
$$F_3(x) = \left\{\, 2^{i_3 x}\right\}.$$

This division into classes provides the foundation of our performance-modeling technique; however, we do not claim that the above classes are exhaustive, and new ones can be added on demand to reflect changes in algorithms and applications. The basic modeling technique will nevertheless be the same.

We first classify the leading-order term of the expectation $E(x)$ according to our scheme. Note that expectations that combine components from different classes $F_k(x)$ are classified according to the component from the highest class. For example, an expectation $\mathcal{O}(p \log p)$ will be classified as belonging to $F_2(x)$. Since we assume that $E(x)$ is sound and our goal is to validate it, we are not interested in a wide deviation limit. Therefore, if the user provides no such limit we choose a default deviation $D(x)$ from the same class. In other words, if $E(x)$ was classified as belonging to $F_k(x)$ we define $D(x)$ by halving the leading-order term exponent i_k of $E(x)$. The lower deviation limit is then defined as $D_l(x) = E(x)/D(x)$, and the upper deviation limit is defined as $D_u(x) = E(x) \cdot D(x)$. By default, the model search space boundaries extend beyond the deviation limits $D_l(x)$ and $D_u(x)$, thus the lower boundary is defined as $B_l(x) = 1$ and the upper boundary as $B_u(x) = E^2(x)$. These boundaries limit the search space of possible models. The deviation limits $D_l(x)$ and $D_u(x)$, on the other hand, define our bug criteria—if a model falls outside these limits, we classify this as a scalability bug. Fig. 3 demonstrates the difference between the search space boundaries and deviation limits. This difference also defines the match criteria. The checkmark ✓ in the figure indicates an exact match between the generated model and the expectation $E(x)$. The approximation sign $\approx$ indicates that the generated model is within the deviation limits, thereby resulting in approximate match. The x sign indicates that the generated model is outside the deviation limits, resulting in no match. The default choice of $D(x)$ ensures a reasonable tolerance for an approximate match. Users that need a lower or higher tolerance can specify a different $D(x)$.

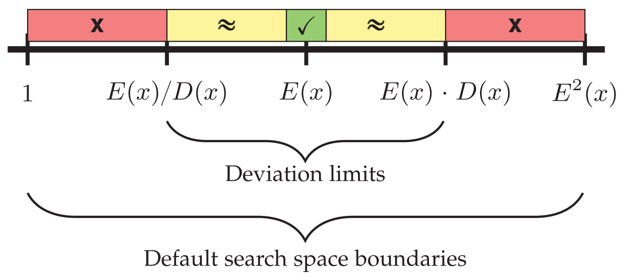

Fig. 3. Search space boundaries and deviation limits relative to the expectation $E(x)$.

The next step is to choose the functions inside the model search space, and thus define its resolution. The user can provide his own search space or let it be generated automatically using the expectation $E(x)$. The construction of the search space is analogous to placing ticks on a ruler. The bigger ticks (e.g., centimeters) are the terms from the class $F_k(x)$ to which $E(x)$ belongs. The outermost ticks are by default B_l and B_u, while the inner ticks are constructed by recursively halving the intervals between existing ticks. The recursion terminates after a defined number of steps, which can be configured before the model generation step. The default, however, is two steps. Each new tick corresponds to a new term and is added to the search space. Practically, this is achieved by averaging the exponents of adjacent terms that are already in the search space. We denote the set of exponents of the terms already in the search space as $I_k \subset \mathbb{Q}$, which means we can define the search space up to this point as $\{f(x) \in F_k(x) \,|\, i_k \in I_k\}$. By introducing smaller ticks (e.g., millimeters), we can increase the resolution even further. In contrast to the bigger ticks, smaller ticks are constructed by multiplying the terms from the class $F_k(x)$ that are already elements of the search space with terms from $F_{k-1}(x)$. As a rule of thumb and a default choice, the first term we select from $F_{k-1}(x)$ has an exponent of 1. We can then expand this selection as needed by incrementing and decrementing the exponent by a step of 1, 1/2, 1/3, and so on. Selecting more terms from $F_{k-1}(x)$ increases the search space resolution, which incurs more overhead and is not always needed. We do not consider any terms from a class lower than $F_{k-1}(x)$ because it would result in ticks that are too fine-grained to characterize significant deviations. Finally, we multiply each term in the search space with a coefficient placeholder a and add another coefficient placeholder c, such that each term $f(x)$ becomes a function $c + a \cdot f(x)$. Both coefficients will be instantiated when fitting the functions in the search space to actual measurements.

We offer both simplicity and flexibility to the user. The only input that the user has to provide is the expectation. The deviation limit and the search space can then be generated automatically, thus relieving the user of the complexity of too many choices. If more flexibility is required the deviation limit and the search space can be modified manually by expanding the default boundaries or increasing the resolution. There is a trade-off, however, between accuracy and speed; therefore, applying these modifications might increase the model-generation time. As an approximate reference, the model generation process in each of our case studies, including all the functions under investigation, never took more than a few seconds.

As an example, let us consider the expectation $E(p) = p$. In this case, the default deviation limit is $D(p) = p^{\frac{1}{2}}$ since it is exactly half of the power of p. The default lower and upper search space boundaries are 1 and p^2, respectively. At this point, our search space is $\{1,\, p, p^2\}$. By averaging the exponents of adjacent terms we construct the models $p^{\frac{1}{2}}$ and $p^{\frac{3}{2}}$, which results in a search space $\{1, p^{\frac{1}{2}},\, p, p^{\frac{3}{2}}, p^2\}$. In the next step, in which we average the exponents of adjacent terms again, we add the terms $\{p^j \mid j = \frac{1}{4}, \frac{3}{4}, \frac{5}{4}, \frac{7}{4}\}$. We then select a term with exponent 1 from the next lower class, $\log p$ in this case, and multiply it by the terms that are already inside the search space. Note that we skip the upper boundary p^2 in order to keep the search space within our defined boundaries:

$$\left\{1,\, \log p,\, p^{\frac{1}{4}}, p^{\frac{1}{4}} \log p,\, p^{\frac{1}{2}},\, \ldots,\, p,\, p \log p,\, \ldots,\, p^{\frac{7}{4}} \log p,\, p^2\right\}.$$

We use the performance-model generator in Extra-P [11], a tool for automated performance modeling of HPC applications. The model generator has already shown to confirm known performance models of real applications as well as discover previously unknown scalability bottlenecks [6], [12], and has also been validated using a wide range of synthetic functions [13]. Furthermore, specific usage examples include modeling the performance of OpenMP constructs [14] and the isoefficiency functions of task-based programs [15]. The generator requires a set of measurements as input whose precise nature depends on the scaling objective (e.g., number of processes versus input size, weak versus strong). As a rule of thumb derived from our experience, the generator needs at least five different settings of the model parameter (e.g., five different numbers of processes). First, it starts by fitting each function in the search space to the measurements using linear regression and leave-one-out cross-validation. In other words, each function in the search space becomes a candidate model. Note that cross-validation allows us to reduce the error when the model is applied to new data. The generator then selects the candidate model with the highest adjusted coefficient of determination $\bar{R}^2$. The coefficient of determination is the ratio of the variation explained by the model to the total variation [16]. In general, a higher value of this coefficient indicates a better fit (the range is [0,1]). The adjusted coefficient of determination penalizes higher numbers of terms in the model, which offers protection against overfitting.

2.4 Validate Expectations

Since we accept expectations in big O notation, we first need to transform the generated models accordingly. This involves isolating the leading-order term in a model and stripping off its coefficient.

Unfortunately, run-to-run variation, which affects almost any system, may introduce a certain degree of noise into the measurement data. This means that we are confronted with a trade-off decision. On the one hand, if we increase the search space resolution, we have to accept that the model would not only reflect the behavior we are interested in but potentially also the noise. On the other hand, if we restrict the resolution too much, we have to accept models that do not fit the data precisely, increasing the likelihood that they

TABLE 1
Performance Expectations of MPI Collective Operations Assuming a Message Size of 800 Bytes and Power-of-Two Number of Processes

Collective	Expectation	Source	Comments
Runtime			
Barrier	$\mathcal{O}(\log p)$	Hensgen et al. [19]	Dissemination
Bcast	$\mathcal{O}(\log p)$	Chan et al. [2]	
Reduce	$\mathcal{O}(\log p)$	Chan et al. [2]	
Allreduce	$\mathcal{O}(\log p)$	Chan et al. [2]	
Gather	$\mathcal{O}(p)$	Chan et al. [2]	
Allgather	$\mathcal{O}(p)$	Chan et al. [2]	
Alltoall	$\mathcal{O}(p \log p)$	Bruck et al. [20]	Indexing alg.
Memory			
Comm_create	$\mathcal{O}(p)$	MPICH [3], [21]	Max grp. size
Comm_dup	$\mathcal{O}(1)$	Balaji et al. [17]	
Win_create	$\mathcal{O}(p)$	MPICH [3], [21]	
Cart_create	$\mathcal{O}(p)$	MPICH [3], [21]	W/reordering
MPI memory	$\mathcal{O}(1)$	Balaji et al. [17]	

will misguide the user. Since according to our experience the latter option is more dangerous, we decided to allow more fine-grained model choices.

To assist the user in understanding the results we define the *divergence model* to be $\delta(x) = G(x)/E(x)$, where $G(x)$ is the generated model and $E(x)$ is the expectation provided by the user. This model characterizes the degree of divergence between the expectation and the observed behavior. It can also be used to visualize the severity of the deviation. Thus, the output we present to the user consists of $G(x)$, $\delta(x)$, and a match rank with three possible indications, as depicted in Fig. 3: total match (meaning $G(x)$ corresponds to $E(x)$), approximate match ($G(x)$ is within the deviation limits), and no match ($G(x)$ is outside the deviation limits).

Severe divergence can either point to a bug in the algorithm, a bug in its implementation, a constraint of the underlying architecture, an unrealistic expectation, or a combination of several factors. The root cause is not always obvious. For example, even if the implementation seems correct at the first glance, it is always possible that bugs, such as false sharing, unnecessary synchronization, or poor communication schedules, increase the actual complexity of the implementation. Nonetheless, the performance litmus test introduced earlier can help separate architectural from implementation constraints. Based on the generated models, we can now also verify the compliance of the actual behavior with the optional cross-function rules. As defined above, cross-functions rules specify relationships between the scaling behavior of different functions. For this purpose, we combine the models involved in such rules before transforming them into their asymptotic form. Finally, if the generated models fall within the deviation limit (i.e., match the expectations either exactly or approximately) the user may instantiate them to predict the scaling limits of selected library functions at specific target scales.

3 Case Study: MPI

MPI is a fundamental building block in most HPC applications, and previous work identified the runtime of collective operations and memory consumption as two potential scalability obstacles [17], [18]. This makes MPI an ideal case study for testing our approach. First, we discuss the framework workflow in the context of MPI, and then we continue with the initial evaluation of MPI collective operations on three different machines, namely a Blue Gene/Q system, an Intel CPU-based cluster, and a Cray XC-30 system. We finish with an evaluation of Intel MPI and Open MPI on a single, Intel CPU-based system.

3.1 Scalability Validation Workflow

We now present the steps of our framework in the context of the MPI case study. The benchmark design is discussed in more detail as it is important to understand how we benchmark and measure our target functions and metrics. This case study can be used as a guideline for applying the test framework to other libraries.

3.1.1 Expectations

The first step in the workflow requires us to choose the metric and the evaluated functions, as well as identify our performance expectations. In this case, we choose to focus on the most common MPI collective functions and their latency-oriented (i.e., small messages) execution times, as well as on the memory overhead of communicators and the resident memory size of an MPI process. Specifically, we look at: *Barrier*, *Bcast*, *Reduce*, *Allreduce*, *Gather*, *Allgather*, and *Alltoall*. By focusing on latency, that is, message sizes of 800 bytes (i.e., 100 doubles), we limit ourselves to only one aspect of performance. It is sufficient for the initial study, but the message size is a changeable parameter and the study could be extended to include bandwidth as well. We measure the memory overhead of communicators by measuring the memory overhead of the *Comm_create*, *Comm_dup*, *Win_create*, and *Cart_create* functions. Lastly, we analyze the resident memory size by estimating the process memory allocated during the benchmark execution.

Table 1 depicts the expectations for the runtime of collective operations in our MPI case study. These expectations come from a number of sources, including studies by Bruck et al. [20] and Chan et al. [2] and MPICH [21], a well-known implementation of MPI from which numerous other implementations are derived. The cost models from these sources incorporate years of research and optimizations that make them a good reference for comparison. Many implementations of MPI collective operations (including MPICH) use different algorithms depending on the message size and the number of processes. Since we use a small message and numbers of processes equal to a power of two, we selected the expected models such that they reflect this setup. The expectations for communicator memory overheads are taken from MPICH and the analysis by Balaji et al. [17], which also points out that a scalable MPI library should consume a fixed amount of memory, independent of the number of processes.

MPI performance guidelines specify internal performance consistency rules between MPI functions [10], [22], [23]. These rules define consistency expectations, and we specifically evaluate two guidelines: *Allreduce* $\preceq$ *Reduce* $+$ *Bcast* and *Allgather* $\preceq$ *Gather* $+$ *Bcast*. These define the cross-function rules that we focus on. The first guideline states that, since semantically it is the same operation, it is reasonable to expect from a correct and optimized MPI implementation that the model for the execution time of `MPI_Allreduce` does not grow faster than the combination of models for the execution time of `MPI_Reduce` and `MPI_Bcast`. Specifically, we combine models using their leading order terms. The same logic also applies to the second guideline.

3.1.2 Benchmark Design

Although the benchmark we designed focuses on MPI, the general structure and principles can be adapted to other libraries as well. It consists of a series of smaller micro-benchmarks that evaluate different collective functions, either in terms of execution time or memory consumption. Each one produces results that are later used as input to the model-generation phase of the framework. It is important to note that contrary to a previous work on automated performance modeling [6], we do not use Scalasca [24] or Score-P [25] in our workflow. The collective operations are benchmarked, but are not instrumented internally. This allows us to use a more suitable mechanism for timing collective operations. To obtain timings for collectives, we adopted the approach by Hoefler et al. [26], which first forces all processes to start the collective operation at the same time, and then finds the maximum runtime across all processes. According to this method, we first calculate clock differences relative to the root process, and then set a time window in the future, relative to the this process, in which every process should start the operation. An earlier version of this window-based technique was suggested in the *SKaMPI* benchmark [27].

Listing 1. Micro-Benchmark Pseudocode for MPI Collectives

```
1:  Perform warm-up runs
2:  **repeat**
3:      $ms_i \leftarrow$ Current memory consumption
4:      Synchronize function start time
5:      $s_i \leftarrow$ Operation's start time in process $i$
6:      Run collective operation
7:      $e_i \leftarrow$ Operation's end time in process $i$
8:      $me_i \leftarrow$ Current memory consumption
9:      Continue to next iteration if synchronization errors
        occurred
10:     $t_j \leftarrow \max_{i=1\ldots P}(e_i - s_i)$
11:     $m_j \leftarrow \max_{i=1\ldots P}(me_i - ms_i)$          $\triangleright$ Memory overhead
12:     Write $t_j$ and $m_j$ to the output
13: **until** $R$ valid runs performed
```

Listing 1 presents the pseudo-code for the micro-benchmark. It starts with a number of warm-up runs and continues to execute the collective operation R times. The warm-up is necessary to eliminate the effects of a cold cache and make sure that the MPI library is fully loaded. Before each run, the window-based technique on line 4 ensures that the collective operation starts approximately at the same time on each process. Even with the most precise synchronization, distributed processes will not be able to start the operation at exactly the same moment in time due to local OS-related noise. However, the window-based synchronization we use eliminates the effect of long delays caused by imbalances in previous computation and communication phases on the timing of collective operations. We measure the memory overhead by wrapping `malloc` and `free` and, for operations that create a new communicator, it gives us exactly the memory overhead of the new communicator. The results of each repetition (both the runtime and the memory overhead) are reduced to a maximum value across all processes.

The number of repetitions R should be high enough to get statistically sound results, especially if the benchmarks are executed in a noisy environment. However, if R is too high it could result in spending too much time benchmarking a single collective operation. This is particularly true for time-consuming collective operations such as `MPI_Alltoall`. As a rule of thumb, we can deem R to be high enough when the 95 percent confidence interval is no larger than 5 percent of the mean.

One way to estimate the resident memory allocated to a process on Linux and Unix-like systems is to analyze the mapped memory regions in the */proc/self/smaps* file. Following this approach, we count either the shared and the private regions, or the proportional set size (PSS) of the process. On Blue Gene/Q the compute nodes run a special minimal version of the Linux kernel that preallocates the memory for the process in advance and does not provide the actual status of the memory in */proc/self/maps*. As an alternative, we use the `Kernel_GetMemorySize` function to obtain the desired value.

To help identify architectural constraints, or negative effects of neighbor network activity, we calibrate the benchmark by running a manually implemented binomial-tree broadcast [2] as our performance litmus test. It is implemented using point-to-point MPI functions and we understand the precise behavior of this implementation under ideal conditions. If its generated performance model does not correspond to the expected analytical model, it suggests that other factors, such as network contention or neighbor activity, are influencing the runtime. After this calibration, we can attribute unexpected behavior with greater confidence to either problematic implementations or to machine-related overheads.

The benchmark runs are orchestrated by the Jülich Benchmarking Environment (JuBE) [28], which allows the user to configure a wide choice of execution parameters and specify ranges for some of them. For example, the user specifies the number of processes per node and a range for the requested nodes. JuBE iterates over the ranges provided by the user independently and creates a batch job for each combination. After the execution is finished it runs optional scripts for results verification and data analysis.

3.1.3 Generation of Scaling Models

The inputs of the model generation phase are runtimes of collective operations, communicator memory overheads, and the estimate of the resident memory size, measured for an increasing number of processes. Many benchmarks reduce the results of multiple iterations to a single value by using an average. In our case, however, to mitigate significant noise we use the first quartile. By choosing this approach, we shift our focus from the average case toward the best case and reduce the risk of false positives that can occur when the levels of noise are very high. At any rate, the divergence model in the average case is as big as in the best case.

As depicted in Table 1, there were four different expectations in this case study: $\mathcal{O}(1)$, $\mathcal{O}(\log p)$, $\mathcal{O}(p)$, and $\mathcal{O}(p \log p)$.

TABLE 2
Machine Specifications for the Evaluation of the Native MPI Implementations (Cores and Memory Size are Given Per Node)

	Juqueen	Juropa	Piz Daint
Platform	Blue Gene/Q	Intel, IB	Cray-XC30
Topology	5D torus	Fat tree	Dragonfly
Nodes	28,672	3,288	5,272
CPU	PPC A2	Xeon X5570	Xeon E5-2670
Clock	1.6 GHz	2.93 GHz	2.6 GHz
Cores	16	8	8
Memory	16 GB	24 GB	32 GB
MPI	PAMI	ParaStation	Cray

The first two were classified as belonging to the class $F_1(x)$, and the other two as belonging to $F_2(x)$. Note that $\mathcal{O}(1)$ is a special case; it can be assigned to any one of the classes by choosing the exponent of 0. The default choice, therefore, is to classify it as belonging to $F_1(x)$. For more consistency, we decided to set the search space in all the cases to the default search space of expectation $\mathcal{O}(p \log p)$. In other words, the search space in all the cases was defined by logarithms with powers of 0 and 1, and powers of $0, \frac{1}{4}, \frac{1}{3}$ and all their multiples up to 2 for p. We also used the same deviation of $\sqrt{p}$ for all the expectations.

3.1.4 Validation of Expectations

In this step, we automatically validate the generated performance models against our expectations. We compute the divergence models and evaluate the cross-function consistency expectations. The final output is a list of generated models, in which each model has an adjusted coefficient of determination, a divergence model, and a match indicator. Table 4 is an example of such a list. The divergence model and the match indicator have already been discussed in Section 2.4.

3.2 Evaluation with Three Native MPI Implementations

In this section, we evaluate our approach on three different machines with a different native MPI implementation on each of them. As already explained, we measured the runtime of collective functions, the memory overhead of communicators, and the memory allocated by the process during the benchmark execution. We first present the experimental setup including machine details and then discuss the results.

3.2.1 Experimental Setup

Table 2 presents the specifications of the three machines on which we conducted our experiments and tested our approach. The first one is Juqueen [29], a Blue Gene/Q machine built by IBM. It is specifically designed for highly scalable codes and features improved energy efficiency. The specialized Compute Node Kernel (CNK) on the compute nodes reduces jitter and allows for reproducible measurements. The second machine, which is based on an Intel architecture, is Juropa. At the time the evaluation was conducted, Juqueen was the capability supercomputer at Forschungszentrum Jülich (FZJ) and Juropa was the capacity machine. The third machine is Piz Daint, an x86-based Cray-XC30 machine at the Swiss National Supercomputing Centre (CSCS). It was built by Cray and therefore has both a different network topology and a different MPI implementation [30]. To enable better scalability and reduce jitter, the compute nodes on Piz Daint run an optimized version of Linux called Compute Node Linux (CNL). At the time the evaluation was conducted, it was the flagship system of CSCS. We believe the differences between these machines make them good choices for our case study and allow us to evaluate the scalability of different MPI implementations.

The MPI implementation on Juqueen is based on the PAMI interface [31] and uses special hardware components to accelerate collective functions [32]. Users have a choice of various protocols for some of the frequently used collective functions, for example, binary or binomial tree for `MPI_Allreduce`. They also have the option to revert to the plain MPICH implementation from which the Blue Gene version was derived. Juqueen provides an extension of MPI that makes it possible to query which algorithm was actually used during the execution of a collective function. The ParaStation MPI on Juropa is based on MPICH as well. It is optimized to select the most appropriate of all available interconnects at runtime. For intra-node communication, for example, it will use shared memory and revert to InfiniBand for inter-node communication [33]. Piz Daint is a Cray machine and uses Cray MPI, which is a vendor implementation of MPI and is quite closely coupled to the machine itself. In these cases, support for non-native implementations, such as Open MPI, is quite limited. Therefore, we chose to focus our initial evaluation on supported implementations, that is, PAMI on Juqueen, ParaStation MPI on Juropa, and CrayMPI on Piz Daint. In a later work that we discuss in Section 3.3, the scalability validation framework was applied to Intel MPI and Open MPI as well. This point is particularly important since MPICH algorithms have known formulae for their execution time, which allows the generated empirical model to be compared to the analytical one. Open-source, PAMI-based algorithms also provide analytical models for their execution time, thus allowing us to get clear expectations about their performance.

Vendor implementations of MPI are quite strongly coupled to the actual machine. Comparing them to a non-native MPI implementation would be unfair since the latter cannot use specialized vendor hardware to run faster. Therefore, instead of evaluating different implementations on the same machine, we looked at native implementations on different machines. It is important to remember that different machines may have different network topologies with varying network latencies and contention points. On Juqueen, the 5D torus topology and the built-in messaging unit (MU) component allow for higher bandwidth and lower latency compared to more conventional 3D torus and fat-tree topologies [32]. Therefore, each native implementation should be considered separately and compared to corresponding analytical models rather than to an implementation on a different machine.

All three machines in our experiments provide highly accurate, high-resolution hardware cycle counters in the form of registers that can be read very quickly with an atomic instruction: MFTB on PowerPC, and RDTSC on x86. All the experiments were performed with a fixed CPU frequency, and to obtain execution time the frequency was multiplied by the cycle count.

TABLE 3
Execution Parameters for the Scalability Validation of MPI Collective Operations

	Juqueen	**Juropa Piz Daint**	**Lichtenberg**
Processes	$\{2^6,\ 2^7,\ \ldots,\ 2^{12}\}$		$\{16,\ 32,\ \ldots,\ 512\}$
Procs. / node	16	8	16
Message size [B]	800	800	800

The table also includes the parameters for the experiments on Lichtenberg (Intel MPI and Open MPI) discussed in Section 3.3.

We note that the experiments on Piz Daint were performed with default Cray MPI library optimizations. Newer versions of Cray MPI have additional algorithms that may improve scaling and can be used by setting appropriate environment variables.

Table 3 presents the execution parameters for the evaluation. Note that the number of MPI processes per node equals the number of cores in the node. The reason is that oversubscribing, namely running more processes than the number of cores, can cause network contention at the node level. On the other hand, undersubscribing, namely having fewer processes, can potentially cause insufficient utilization of the node's computational resources. This is because the adoption of multithreaded programming models is neither ubiquitous among HPC applications nor can every application readily benefit from multithreading.

3.2.2 Analysis of the Results

Tables 4 and 5 present the results of our analysis. Both tables show the generated models next to our expectations. Table 4 refers to runtime and Table 5 to memory metrics. Since the size of the memory growth coefficients may be significant, we show full models of memory overheads and estimated memory consumption by MPI. The $\bar{R}^2$ row lists the adjusted coefficient of determination, which indicates how well the data fits a statistical model. It is used in the model generation phase to create models that fit the data better [6]. Note that $\bar{R}^2$ is not applicable to constant models. Following $\bar{R}^2$ is the row with the divergence models δ as defined in Section 2.4. Finally, the match row specifies whether the generated model meets our expectations. If the two are in agreement, a checkmark ✓ is shown. If the match is approximate according to the definition in Section 2.4, an approximation sign $\approx$ is shown. A solid **x** represents an unquestionable mismatch. A warning sign ⚠ indicates the violation of a performance guideline. Fig. 4 depicts the runtime models for the collective functions we benchmarked. The circles, squares, and triangles depict the actual measurements, whereas the lines are the predictions. Each curve is annotated with the corresponding model that sits on top of the curve. Since we focus on the scalability behavior of the models, we chose not to show the constant terms. The discussion

TABLE 5
Generated (Empirical) Models of Memory Overhead on Juqueen, Juropa, and Piz Daint Alongside Their Theoretical Expectations

	Juqueen	**Juropa**	**Piz Daint**
MPI memory [MB]			Expectation: $\mathcal{O}(\log p)$
Model	$10.7 \cdot 10^{-3} \cdot \log p$	$16 + 0.56 \cdot p$	$46 + 1.35 \cdot \log p$
$\bar{R}^2$	0.72	1	0.23
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(p/\log p)$	$\mathcal{O}(1)$
Match	✓	**x**	✓
Comm_create [B]			Expectation: $\mathcal{O}(p)$
Model	$2.2 \cdot 10^5 + 24 \cdot p$	$264 + 28 \cdot p$	$3770 + 46 \cdot p$
$\bar{R}^2$	1	1	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Match	✓	✓	✓
Comm_dup [B]			Expectation: $\mathcal{O}(1)$
Model	$2.2 \cdot 10^5$	256	$3770 + 18 \cdot p$
$\bar{R}^2$	-	-	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$
Match	✓	✓	**x**
Win_create [B]			Expectation: $\mathcal{O}(p)$
Model	$96 \cdot p$	$256 + 60 \cdot p$	$3287 + 118 \cdot p$
$\bar{R}^2$	1	1	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Match	✓	✓	✓
Cart_create [B]			Expectation: $\mathcal{O}(p)$
Model	$2.2 \cdot 10^5 + 52 \cdot p$	$356 + 24 \cdot p$	$2545 + 63 \cdot p$
$\bar{R}^2$	0.99	1	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Match	✓	✓	✓

TABLE 4
Generated (Empirical) Runtime Models of MPI Collective Operations on Juqueen, Juropa, and Piz Daint Alongside Their Theoretical Expectations

	Barrier	**Bcast**	**Reduce**	**Allreduce**	**Gather**	**Allgather**	**Alltoall**	**Bcast (BT)**
Expectation	$\mathcal{O}(\log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p \log p)$	$\mathcal{O}(\log p)$
Juqueen								
Model	$\mathcal{O}(\log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(\log p)$
$\bar{R}^2$	0.99	0.86	0.93	0.87	0.99	0.99	0.99	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1/\log p)$	$\mathcal{O}(1)$
Match	✓	✓	✓	✓	✓	✓	$\approx$	✓
Juropa								
Model	$\mathcal{O}(p^{0.67} \log p)$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(\sqrt{p} \log p)$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p^{1.25})$	$\mathcal{O}(p^{1.25} \log p)$
$\bar{R}^2$	0.99	0.98	0.99	0.99	0.99	0.98	0.99	0.99
$\delta(p)$	$\mathcal{O}(p^{0.67})$	$\mathcal{O}(\sqrt{p}/\log p)$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(\sqrt{p}/\log p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{0.25}/\log p)$	$\mathcal{O}(p^{1.25})$
Match	**x**	$\approx$	$\approx$	$\approx$	✓	✓	$\approx$	**x**
Piz Daint								
Model	$\mathcal{O}(p^{0.33})$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(\sqrt{p} \log p)$	$\mathcal{O}(p^{0.67} \log p)$	$\mathcal{O}(p)$	$\mathcal{O}(p^{1.25})$	$\mathcal{O}(p^{1.33})$	$\mathcal{O}(p \log p)$
$\bar{R}^2$	0.99	0.94	0.94	0.99	0.99	0.99	0.99	0.99
$\delta(p)$	$\mathcal{O}(p^{0.33}/\log p)$	$\mathcal{O}(\sqrt{p}/\log p)$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(p^{0.67})$	$\mathcal{O}(1)$	$\mathcal{O}(p^{0.25})$	$\mathcal{O}(p^{0.33}/\log p)$	$\mathcal{O}(p)$
Match	$\approx$	$\approx$	$\approx$	**x** ⚠	✓	$\approx$	$\approx$	**x**

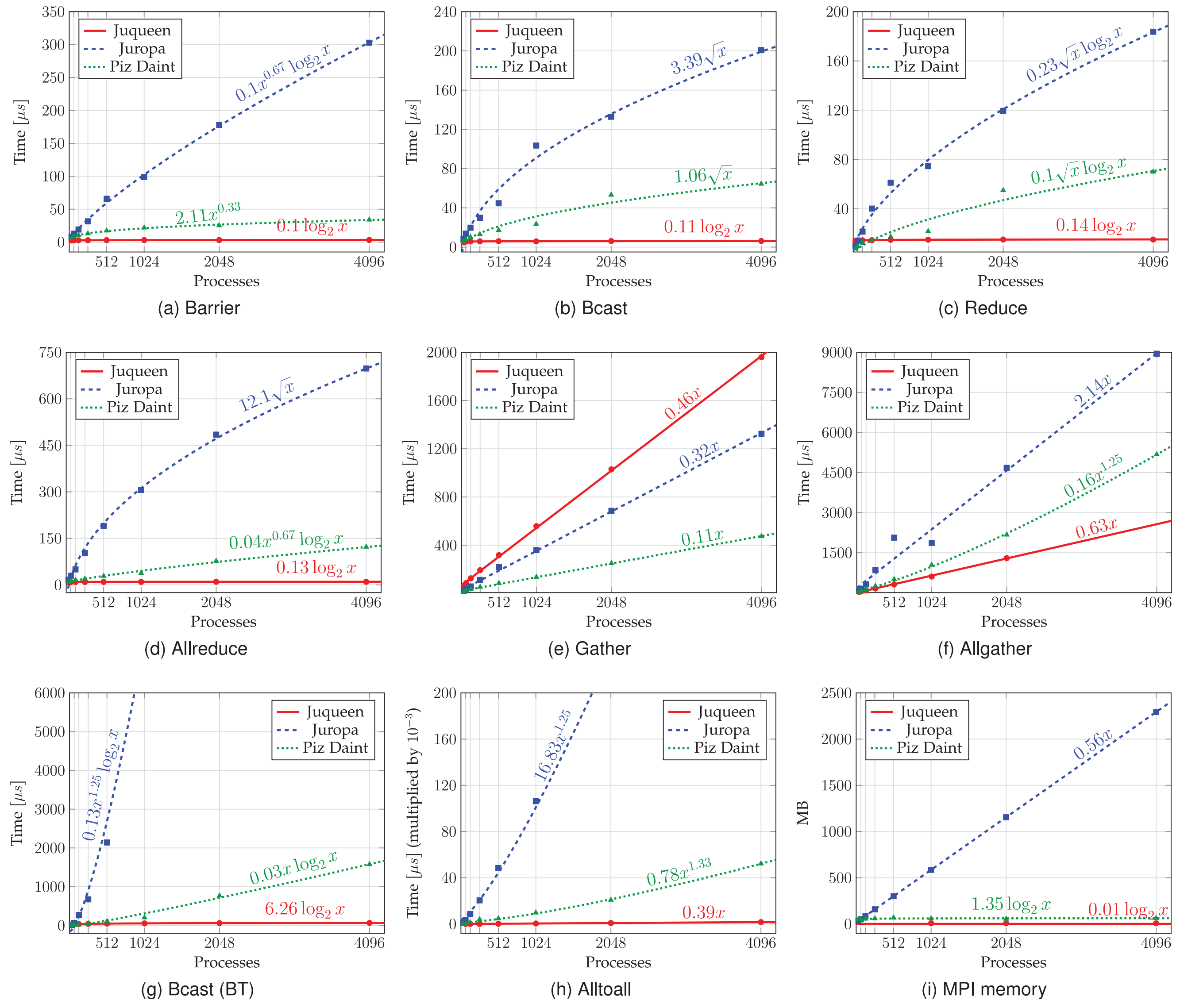

Fig. 4. Measurements (circles, squares, triangles) and generated runtime models (plot lines) on Juqueen, Juropa, and Piz Daint.

below starts with Juqueen, on which almost all the generated models correspond almost fully to expectations. We then continue with Juropa and Piz Daint, on which the results differed from our expectations to some degree.

Juqueen. On Juqueen, the performance of collective functions was generally better than on the other machines and we found that almost all of our expectations were met. All the models on Juqueen are either logarithmic or linear with respect to the number of processes p. As can be seen in Table 4, all the generated models on Juqueen correspond exactly to the expected models with the exception of `MPI_Alltoall`, which is identified as linear when, in fact, the expectation would be $\mathcal{O}(p \log p)$. The difference between reality and expectation is small enough to be explained by noise and other system effects. The manually implemented binomial-tree (BT) version of the broadcast is shown in the rightmost column of Table 4. The expected cost of this algorithm for small messages is: $(\alpha + \beta) \log p$, where α is the message startup time and β is the transmission time per data element; and though it is slower in absolute terms than the native `MPI_Bcast`, the generated model is still logarithmic. Table 5 presents the models for the communicator memory overheads and the estimated fraction of the memory allocated by the process that is consumed by MPI. Although the generated models on Juqueen correspond to the expectations, the linear growth of some of the communicator constructors can still become an issue at very large scale.

Juropa and Piz Daint. On Juropa and Piz Daint, the predicted performance models of some collective functions do not fully match their expectations. These discrepancies between predicted and expected behavior suggest potential scalability issues. Almost all the generated models, including the ones for `MPI_Barrier`, `MPI_Bcast`, and `MPI_Reduce`, do not correspond to the expected logarithmic models. The generated model of the binomial-tree broadcast falls outside the deviation limits and clearly fails to match the expected logarithmic model, too. Since we have a clear understanding of this algorithm and its complexity, we can point to a number of external factors as potential causes of this discrepancy:

1) The network model that was used to calculate the expected cost of the binomial-tree broadcast algorithm

is a simplistic abstraction of a real-world network such as the IB fat-tree interconnect on Juropa.

2) Network hardware and topology can influence the runtime of various collective functions and make them slower than expected [34], [35].
3) On some machines, the performance of applications that use communication extensively strongly depends on the node allocation they receive and the neighborhood of each node [36]. An application that runs on a neighbor node and produces heavy network load creates more perturbation for our benchmark.
4) System noise and jitter could potentially be significant factors that influence the performance [37], [38]. These factors mostly affect Juropa, since it does not have a specialized kernel that has been optimized for noise reduction.

The performance models of `MPI_Gather` on both Juropa and Piz Daint, as well as the `MPI_Allgather` model on Juropa, are linear as expected. On Piz Daint, however, the performance model of the latter does not match the expectation, but still falls within the deviation limits. In Table 4, the warning sign under *Match* signals that a performance guideline violation was detected. As discussed in Section 3.1, the automatic validation evaluates two performance guidelines, one for *Allreduce* and one for *Allgather*. Although the actual measurements on Piz Daint do not violate the *Allreduce* guideline, the generated models predict that the guideline might be violated at larger scales. Note that a performance guideline violation does not imply whether there is a mismatch or an approximate match to the expectation.

The communicator memory overheads on Juropa and Piz Daint are presented in Table 5. On Juropa, the generated models correspond to expectations, and it is interesting to note that the initial overheads (the constants) are very small. This is in direct contrast to Juqueen, on which these constants are much higher. The model for communicator duplication on Piz Daint is linear, although it is expected to be constant. The development team at Cray confirmed that the implementation of `MPI_Comm_dup` was taken from MPICH 3.1.2. We then inspected the source code of this MPICH version and discovered that `MPI_Comm_dup` creates an internal communicator that includes representative processes (masters) from each node. This behavior is also discussed in an MPICH memory usage study [18]. It is clear that this result points to a scalability bottleneck in this operation.

Fig. 4i presents the models for the resident memory size of an MPI process on all three machines. In the case of Juropa, the generated model reveals a severe scalability problem. Even with smaller values of p, it is non-scalable. Starting with 1024 nodes, it is impossible to have 8 MPI processes per node since all the processes would require 35 GB in total and the node's memory is just 24 GB. Our experiments confirmed this memory wall: memory allocation failed when the total number of processes was 8192 (with 8 processes per node). Our findings are confirmed by the documentation; the reason for the linear increase in allocated memory is that ParaStation MPI uses by default the Reliable Connected (RC) InfiniBand service, which needs 0.55 MB of memory for each MPI connection [33]. When using `MPI_Alltoall` each process will allocate $0.55p$ MB of memory, which is exactly the linear behavior we discovered through our scalability validation framework.

3.3 Intel MPI and Open MPI

This section presents an evaluation of two additional implementations of MPI collective operations [39], namely Intel MPI and Open MPI. We follow the same workflow as described in Section 3.1, but used a different benchmark (step 2), which will be described in more detail below. Unlike the initial evaluation that used three different machines, we evaluate Intel MPI 2017 update 1 and Open MPI 2.0.1 on the same system, that is, the Lichtenberg cluster at the Technische Universität Darmstadt [40]. No other MPI implementations (e.g., MPICH) are officially supported on this system.

The first step in the workflow, in which expectations are defined, stays mostly the same. The focus is on the same latency-oriented execution time, but without memory overhead of communicator-related functions and MPI memory consumption. We chose to evaluate the same set of MPI collective operations, that is *Barrier*, *Bcast*, *Reduce*, *Allreduce*, *Gather*, *Allgather*, and *Alltoall*, so the expectations in Table 1 still apply. The third and the fourth step, namely the generation of models and the validation of expectations, respectively, stay the same as well.

3.3.1 Benchmark Design

Following recent studies on MPI benchmarking accuracy [41], [42], we used the ReproMPI benchmark [43] rather than the benchmark suggested earlier in Section 3.1. Although the basic structure of ReproMPI resembles our earlier benchmark, it features an improved version of the window-based synchronization technique, as well as a flexible mechanism for predicting the number of repetitions that are required to obtain statistically sound results. We configured ReproMPI to use the same timing mechanism suggested earlier, namely the RDTSC register available on x86 platforms and a fixed clock frequency. By default, ReproMPI outputs the runtime of an operation for each process. To be consistent with the benchmark in Section 3.1, an additional script was used to find the maximum result across all the processes.

3.3.2 Experimental Setup

The benchmarks were executed on a separate island of 32 nodes on Lichtenberg. Each node comprises two Intel Xeon E5-2670 processors with 8 cores (hyper-threading disabled) and 16 GB of memory, which means 16 cores and 32 GB of memory per node. Table 3 presents the input parameters for our evaluation. We used the same configuration of one MPI process per core as before and the same message size of 800 bytes. Following the observations in Section 3.2, the whole island was reserved for each benchmark run. This ensured that no other program used resources within this island and the negative effects of a busy neighborhood [36] were eliminated.

3.3.3 Analysis of the Results

Table 6 shows the evaluation results. The rows follow the same format as in Table 4. Note that `MPI_Gather` is missing from the table because the results indicated that both Intel

TABLE 6
Generated (Empirical) Runtime Models of Intel MPI and Open MPI Collective Operations Alongside Their Theoretical Expectations

Expectation	**Barrier** $\mathcal{O}(\log p)$	**Bcast** $\mathcal{O}(\log p)$	**Reduce** $\mathcal{O}(\log p)$	**Allreduce** $\mathcal{O}(\log p)$	**Allgather** $\mathcal{O}(p)$	**Alltoall** $\mathcal{O}(p \log p)$	**Bcast (BT)** $\mathcal{O}(\log p)$
Intel MPI							
Model	$\mathcal{O}(p)$	$\mathcal{O}(p^{0.75} \log^2 p)$	$\mathcal{O}(p^{0.75} \log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(p^{2.75})$	$\mathcal{O}(p \log p)$	$\mathcal{O}(\log p)$
$\bar{R}^2$	0.99	0.99	0.97	0.74	0.99	0.99	0.92
$\delta(p)$	$\mathcal{O}(p/\log p)$	$\mathcal{O}(p^{0.75} \log p)$	$\mathcal{O}(p^{0.75})$	$\mathcal{O}(1)$	$\mathcal{O}(p^{1.75})$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Match	x	x	x	✓	x	✓	✓
Open MPI							
Model	$\mathcal{O}(\log^2 p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(\log^2 p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(p \log^2 p)$	$\mathcal{O}(p \log p)$	$\mathcal{O}(\log^2 p)$
$\bar{R}^2$	0.99	0.99	0.99	0.99	0.99	0.99	0.99
$\delta(p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(1)$	$\mathcal{O}(\log p)$	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 p)$	$\mathcal{O}(1)$	$\mathcal{O}(\log p)$
Match	≈	✓	≈	✓	≈	✓	≈

MPI and Open MPI changed the underlying algorithm of this operation when the number of processes was increased. At the time of the study, we were unable to find a way to disable the algorithm switch. This is an example for a use case for the *segmented modeling* approach [44] that aims to solve the problem where the algorithm changes its behavior substantially for some range of the input parameter. A substantial change means that a different model is needed to explain the new behavior. In other words, the measurements cannot fit accurately just one model, hence, we need to find an inflection point or possibly a number of inflection points and fit a different model for each segment between these points. The segmented modeling technique tries different potential inflection points and checks whether the two new models give us a better fit. Naturally, this approach requires using more values for the input parameter, since each segment is smaller than the whole range of values we have. In our case, however, the maximum number of processes per island is 512, which means increasing the number of processes would have required using two separate islands and this would have exacerbated the influence of the network topology on the measured runtime [34], [35].

Intel MPI. In the case of Intel MPI, about half of all the the predicted performance models do not match the expectations. `MPI_Bcast` and `MPI_Allgather` are particularly problematic. Figs. 5b and 5e demonstrate that the models for these operations grow much faster than the corresponding models for Open MPI, which are closer to the expectations. The last column of Table 6 shows the Intel MPI model for the binomial-tree version of the broadcast operation. The implementation of this litmus test is based on MPI point-to-point communication, and since the benchmarks were performed on a separate, exclusively reserved island, the results clearly point to potential implementation issues in Intel MPI. In the case of `MPI_Barrier` and `MPI_Reduce`, Figs. 5a and 5c show that the execution times of Intel MPI are better or on par

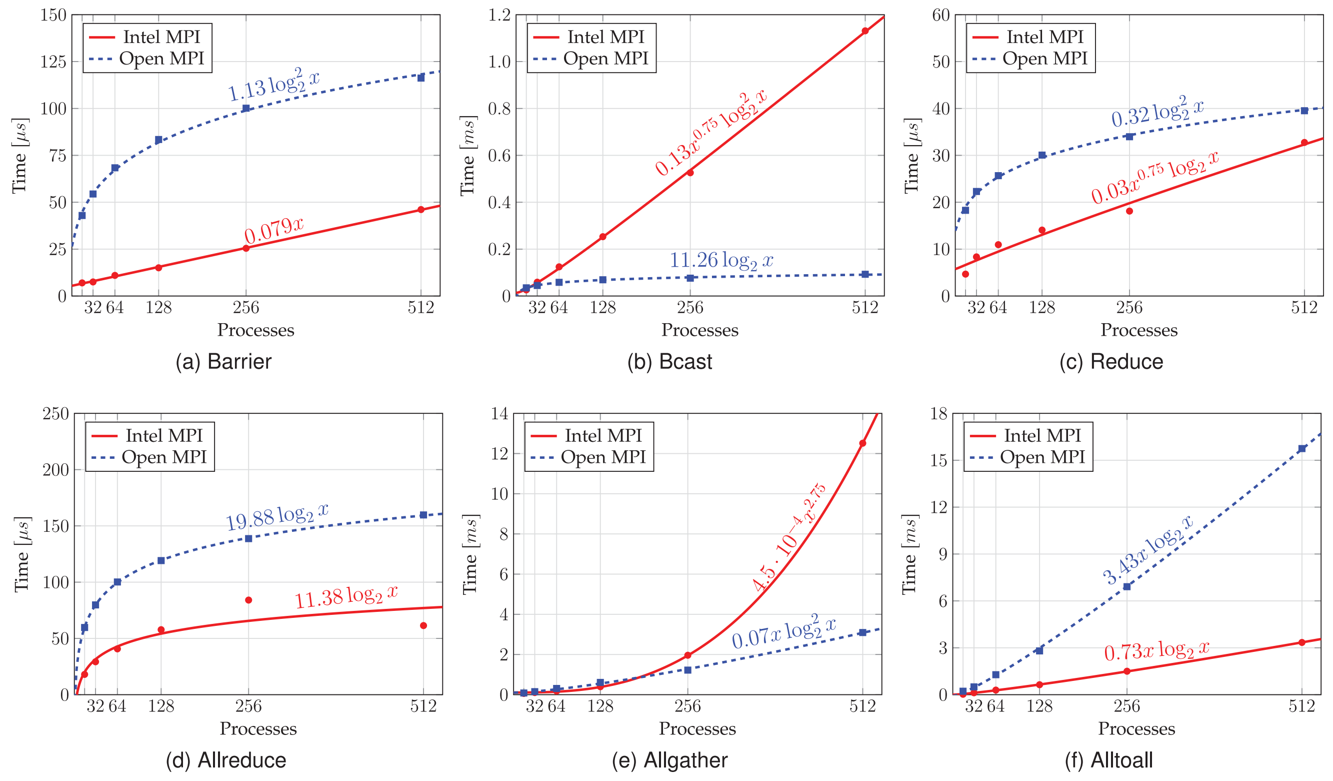

Fig. 5. Measurements (circles, squares) and generated runtime models (plot lines) of some of the collective operations in Intel MPI and Open MPI.

TABLE 7
Generated (Empirical) Runtime Models of MAFIA Functions Alongside Their Theoretical Expectations

	gen	dedup	pcount	unjoin
Expectation	$\mathcal{O}(k^3 2^k)$	$\mathcal{O}(k^4 2^k)$	$\mathcal{O}(k 2^k)$	$\mathcal{O}(k^3 2^k)$
Model	$\mathcal{O}(k^4 2^k)$	$\mathcal{O}(k^4 2^k)$	$\mathcal{O}(k 2^k)$	$\mathcal{O}(k^2 2^k)$
$\delta(k)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1/k)$
Match	$\approx$	✓	✓	$\approx$

with Open MPI. Furthermore, the predicted models have relatively small coefficients. This means that the mismatch in this case is most likely caused by OS jitter and noise in general, rather than by implementation issues.

Open MPI. In the case of Open MPI, there are no mismatches at all and almost half of the models correspond to the expectations. Table 6 shows that the model for the binomial-tree version of the broadcast operation is slightly worse than expected. Since the benchmarks were executed on a separate island of Lichtenberg, network interference was not a factor in this case. Lichtenberg nodes do not have a specialized kernel, thus the likely cause of discrepancies is OS jitter [38].

From the Intel MPI and Open MPI results it is not possible to derive any conclusion as to how well either of these MPI implementations perform relative to MPICH, which is the basis for the MPI implementations in Section 3.2. The reason is that the evaluation was performed on different machines and under different conditions (e.g., unique networking hardware on Juqueen and separate island of 32 nodes on Lichtenberg).

4 Case Study: MAFIA

No matter how large the degree of parallelism, optimizing sequential code is still essential to achieve good performance. The subject of our next case is therefore Merging of Adaptive Finite IntervAls, a sequential data-mining program. One of the basic problems in data mining is identifying regions of similarity in a multi-dimensional data set. Many applications, however, exhibit a high degree of dimensionality in the data, which makes traditional approaches of all-attribute clustering problematic. A possible solution is to use subspace clustering methods to identify clusters in a subset of dimensions. MAFIA is one example of such a method. It is a serial algorithm for subspace clustering based on adaptive grid methods [8]. The cluster dimensionality k is a critical parameter in this algorithm since the ultimate goal is to identify clusters across all dimensions. Users of MAFIA will start with a smaller k but will be interested in increasing it to catch all the dimensions. We are interested in applying our framework to see whether the scaling expectations as a function of k are valid. This use case is an example of algorithmic modeling since the model parameter k is a parameter of the algorithm itself.

Following the four steps of our approach, we start by defining the expectations. Along with k, the parameters of MAFIA are the number of data points n, the dimensionality of the points d, and the number of clusters m. We further identify four main functions (i.e., kernels) in the main computation phase of MAFIA: (i) *gen*—generation of candidate subsets; (ii) *dedup*—de-duplication of these subsets; (iii) *pcount*—identification of dense subsets (i.e., clusters); and (iv) *unjoin*—determination whether lower dimensional subsets were not already absorbed by the higher ones [45].

Table 7 presents the expectations for these functions provided by Adinetz et al. [8] in their effort to optimize MAFIA. In contrast to the MPI study, all the expectations are exponential: $\mathcal{O}(k2^k)$, $\mathcal{O}(k^3 2^k)$, and $\mathcal{O}(k^4 2^k)$. The focus in this use case was the runtime of the algorithm as k increases; therefore we set the other parameters as follows: $n = 10^5$, $d = 20$, and $m = 3$. The benchmarking process was much simpler in this case since MAFIA is a serial code and we were not modeling scalability on an increasing number of cores. In other words, the experiments were conducted on one node of Juropa and repeated for $k = 3, 4, \ldots, 16$. In all of these experiments we did not change the default deviation limits or the search space boundaries. As Table 7 shows, all the generated models match our expectations completely or are inside the deviation limits. This example illustrates the flexibility of our approach, which can be adapted to different scalability problems with different expectations.

5 Case Study: Parallel Sorting

In this section, we present another use case for the scalability validation framework, namely validation of performance expectations of parallel sorting algorithms. In general, parallel sorting focuses on techniques to solve the sorting problem using parallel processing [46], [47], [48]. Sorting is a fundamental problem in computer science and has many uses. However, with the increasing scale of HPC systems the problems scientists and engineers focus on increase in scale as well. In other words, the input for a sorting algorithm no longer fits into one node and we need, therefore, to formulate the parallel sorting problem [49] in terms of distributed memory:

Input: A distributed sequence of $n = \sum_{i=0}^{p-1} |N_i|$ elements such that each block of elements N_i has $\frac{n}{p}$ elements and is assigned to process p_i.

Output: A permutation of the distributed input sequence such that each process p_i has a block N'_i, in which all elements are sorted, and $\cup_{i=0}^{p-1} N'_i = \cup_{i=0}^{p-1} N_i$. Moreover, for every $i \leq j$ we have $N'_i \leq N'_j$, where $N_i \leq N_j$ means that every element of N_i is less than or equal to every element in N_j. Note that the blocks do not have to be balanced, that is, equal in size across all processes.

Our focus in this use case is on five parallel sorting algorithms: (i) Sample sort [49]; (ii) Histogram sort [47]; (iii) Exact-splitting sort [46]; (iv) Radix sort [50]; and (v) Mini sort [51]. The first three are so called *splitter-based* algorithms. The fourth is a parallel variant of the well-known sequential Radix sort [52], and the fifth algorithm addresses a special case of the parallel sorting problem where $\frac{n}{p} = 1$. For the evaluation, we developed a library with implementations of these algorithms called *libparsort* [53]. As a starting point, we used existing implementations of Sample sort, Exact-splitting sort, Radix sort, and Mini sort created by Elmar Peise and Christian Siebert. We then refactored these implementations and implemented Histogram sort from scratch, as well as added a number of improvements to the Exact-splitting and Mini sort implementations. Note that since we studied the available implementations in detail as

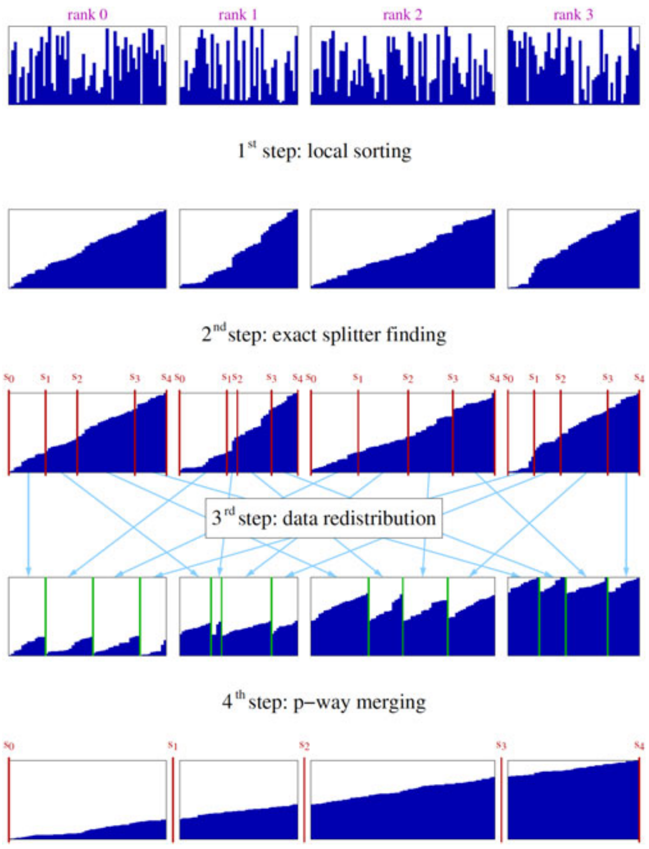

Fig. 6. Parallel sorting algorithm based on finding exact splitters (taken from Siebert and Wolf [46]).

well as developed our own code, there was no need for a performance litmus test as in the MPI case study.

Solutions that try to gather too many elements in one node or that fail to exploit the available parallel resources will not scale. Splitter-based algorithms address these two issues by first letting processes sort their part of the input and then by solving the merging-redistribution problem without relying on any one process in particular. Fig. 6 shows the steps of Exact-splitting sort [46], a variant of a splitter-based algorithm. These algorithms have a common scheme of four steps, namely, sorting the elements locally, finding $p+1$ splitters (the first and the last splitter are implicit) , redistributing the elements according to the splitters, such that all the elements between splitter s_i and s_{i+1} end up in process p_i, and finally, merging all the parts in process p_i locally. The main differences between the splitter-based algorithms is in the order of the solution steps and the technique to find the splitters. A simple variant of a splitter-based algorithm is Sample sort [49]. In this algorithm, each process i selects a sample of $p-1$ candidates from N_i, which it sorts beforehand, and sends them to the root process. The root then sorts these candidates and selects $p-1$ splitters. Eventually it broadcasts the splitters back to the processes. As p increases, gathering and sorting the splitter candidates in the root process will become a significant bottleneck. Histogram sort circumvents this bottleneck by selecting a random sample of splitter candidates across the whole range of the input data. It then computes the prefix sum of local histograms based on the sample. The prefix sum produces the location of each candidate within the eventual sorted array. This allows the algorithm to check whether a candidate falls within the range of an ideal splitter (splitter i location is $\frac{n}{p}(i+1)$). Histogram sort repeats this step until all the splitters are found. The range is a parameter of the algorithm, but by keeping it reasonably small, blocks N_i' will have roughly the same size in the end [47]. The Exact-splitting algorithm aims to find the ideal splitters as well, but instead of relying on histograms it uses an efficient scheme for approximating global medians. By repeating this scheme for every splitter, it guarantees that we find exactly the ideal splitters, in other words, $N_i' = \frac{n}{p}$ for each process. Although finding better splitters comes at the cost of more communication steps, both Histogram and Exact-splitting sort eliminate the bottleneck in the Sample sort algorithm.

Radix sort is not a comparison-based sorting algorithm. Instead it takes advantage of the binary representation of the keys. The idea is to break the keys into digits of one or more bits and then sort one digit at a time. To work properly, the algorithm proceeds from the least significant to the most significant bit using a stable sorting algorithm for each digit. In most cases, Counting sort [52] is used to sort the digits, since it is stable and well suited for this type of input. The parallel variant of Radix sort parallelizes the Counting sort by efficiently counting in parallel using collective reduction operations in MPI.

Mini sort assumes that each process has just one element of input data, that is $n = p$. The idea behind it is similar to Quicksort. Specifically, it uses an efficient scheme to approximate the global median among a group of processes, which is then used as a pivot to partition the processes into three groups: the ones with a smaller value than the pivot, the ones with an equal value, and the ones with a higher value. After exchanging the data elements, the sorting continues recursively in the first and third partitions and stops when a partition contains just one element.

Initial implementations of Exact-splitting sort and Mini sort approximated global medians based on a ternary tree selection proposed by Rousseeuw and Bassett [54]. However, for more robustness and accuracy, we changed the approximation method to a binary tree-based technique proposed by Axtmann and Sanders [48].

TABLE 8
Expected Runtime Complexities of Parallel Sorting Algorithms

Algorithm	Expected complexity
Sample sort	$\mathcal{O}(\frac{n}{p}\log\frac{n}{p} + p^2\log p)$
Histogram sort	$\mathcal{O}(\frac{n}{p}\log\frac{n}{p} + rp\log\frac{n}{p})$
Exact-splitting sort	$\mathcal{O}(\frac{n}{p}\log\frac{n}{p} + p\log^2 n)$
Radix sort	$\mathcal{O}(\frac{b}{k}(\frac{n}{p} + 2^k + \log p))$
Mini sort	$\mathcal{O}(\log^2 p)$

5.1 Expectations

The first step in the scalability validation workflow requires us to choose the metric and identify the performance expectations of the algorithms. In this case, the metric is execution time and the generated models are functions of the number of MPI processes p. Table 8 presents the theoretical runtime complexities of the sorting algorithms [46], [49], [51], [52]. Note that the three splitter-based algorithms have a common component $\mathcal{O}(\frac{n}{p}\log\frac{n}{p})$ for the local sorting step, but

TABLE 9
Execution Parameters for the Scalability Validation of Parallel Sorting Algorithms

	Lichtenberg	Juqueen
Sample, **Exact-splitting** ($\frac{n}{p} = 10^7$, Uniform)		
Processes (p)	$\{32,\ 64,\ \ldots,\ 512\}$	$\{2^{11},\ 2^{12},\ \ldots,\ 2^{16}\}$
Histogram ($\frac{n}{p} = 10^7$, Uniform, Gaussian)		
Processes (p)	$\{32,\ 64,\ \ldots,\ 512\}$	$\{2^{11},\ 2^{12},\ \ldots,\ 2^{16}\}$
Radix, $\boldsymbol{k = 4}$ ($\frac{n}{p} = 10^7$, Uniform)		
Processes (p)	$\{32,\ 64,\ \ldots,\ 512\}$	$\{2^{11},\ 2^{12},\ \ldots,\ 2^{16}\}$
Radix, $\boldsymbol{k = 8}$ ($\frac{n}{p} = 10^7$, Uniform)		
Processes (p)	$\{32,\ 64,\ \ldots,\ 512\}$	$\{2^{9},\ 2^{10},\ \ldots,\ 2^{14}\}$
Mini ($\frac{n}{p} = 1$, Uniform)		
Processes (p)	$\{32,\ 64,\ \ldots,\ 512\}$	$\{2^{11},\ 2^{12},\ \ldots,\ 2^{16}\}$

Input elements are 64-bit integers and there are 16 processes per node in all cases.

they differ in their approaches for finding the splitters. In the case of the Histogram sort, r is the number of repetitions required for the splitter finding phase to converge. Radix sort assumes that the input values are integers of b bits and it breaks these integers into digits of k bits. There are $\frac{b}{k}$ digits and for each digit it runs a parallelized Counting sort [52]. Mini sort works with minimal data (i.e., $n = p$) and therefore its complexity is based only on p.

5.2 Benchmarking Approach

The second step in the scalability validation workflow instructs us to define the benchmarking approach. Previous case studies adopted custom solutions to instrumentation that allowed us to adhere to certain constraints and highlight specific aspects of performance (e.g., MPI collective functions had to start at the same time). In the parallel sorting case, however, we can adopt a more standard approach and use an existing instrumentation solution. Since our model generation tool Extra-P [11] provides direct support for Score-P [25], we used this instrumentation platform to instrument the sorting functions in the library. One advantage of Score-P is that it measures execution times of the whole call tree, such that we obtain not only the execution time of the sorting function itself, but also the times of the functions called within that function. In other words, we can get models for every logical step of the algorithms (e.g., local sorting, splitter finding, and so on).

We ran the benchmarks on two systems, the first one is Lichtenberg (see Section 3.3) and the second one is Juqueen (see Section 3.2). Table 9 presents the input parameters for the evaluation. The input elements were 64-bit integers generated randomly in uniform distribution. For Histogram sort, we also used the Gaussian distribution. The number of elements per process was set to be constant with $\frac{n}{p} = 10^7$ (except for Mini sort with $\frac{n}{p} = 1$), which ensured that we modeled only the influence of p on the performance. The choice of input sizes is based on a previous study of parallel algorithms [48].

5.3 Analysis of the Results

Table 10 presents the results of the evaluation. The expectations are leading order terms of simplified expressions of the runtime complexities from Table 8. Since $n = Cp$, the expression $\mathcal{O}(\frac{n}{p}\ \log \frac{n}{p})$ turns into a constant. In Histogram sort, r depends on the distribution of the input data. We ran the evaluation both with uniform and Gaussian distributions and in both cases $r = 2$. Therefore, we assume that r is constant in the expectation. In Radix sort, b and k do not change during the benchmarking and are considered constant, which means the expectation turns into $\mathcal{O}(\log p)$. However, depending on the value of k, the hidden constant coefficient could become quite large.

The models for Sample sort on both Lichtenberg and Juqueen match the expectation and reflect the splitter-finding complexity. Although on Lichtenberg the execution time of the splitter-finding step was smaller than that of other steps, the model for this step is $\mathcal{O}(p^2\ \log p)$ and dominates the other steps.

The models for Histogram sort reflect both the evaluation with uniform and Gaussian distributions. In both cases, the splitter-finding step converged after two iterations, which means the models for both distributions are the same. The model for Lichtenberg matches the expectation, whereas the model for Juqueen is actually better than the expectation.

In the case of Exact-splitting sort, the model for Lichtenberg does not match the expectation exactly, whereas the model for Juqueen is an exact match. The reason is that Exact-splitting sort requires more communication steps to find the splitters. Specifically, it runs the global approximation step, which uses collective operations, for every splitter (i.e., p times). This means it invokes a large number of collective operations in the splitter-finding step and any potential overheads or inefficiencies are accumulated. Since Juqueen has highly optimized collectives, the accumulated overhead is smaller compared to Lichtenberg and this results in a performance model that matches the expectation.

We evaluated Radix sort with two different digit sizes, namely $k = 4$ and $k = 8$. Since the input values were 64-bit integers (i.e., $b = 64$), the number of the Counting sort steps (i.e., $\frac{b}{k}$) was 16 and 8, respectively. A higher number of Counting sort steps is translated into a larger constant coefficient in the expectation. Table 10 shows that for $k = 4$ the Juqueen model does not correspond to the expectation, while the Lichtenberg model matches only approximately. One possible explanation lies within the implementation of the Counting sort step, which is based on MPI point-to-point communication. Counting sort determines the designated locations of the digits in the global sorted array and uses point-to-point communication to exchange the digits between the processes. Since there were much more processes on Juqueen than on Lichtenberg, the cost of this communication is higher on Juqueen and this is reflected in the generated model. For $k = 8$, the number of Counting sort steps was twice as small leading to a reduced number of point-to-point operations. Besides, the memory consumption of our implementation of the Counting sort step increases with k and p. This forced us to restrict the process counts on Juqueen for $k = 8$ and use up to $p = 2^{14}$ processes. A smaller number of processes reduces the cost of the point-to-point communication, which results in models that match the expectations more closely.

The model for Mini sort on Lichtenberg does not match the expectation, whereas on Juqueen there is an exact

TABLE 10
Generated (Empirical) Runtime Models of Five Parallel Sorting Algorithms: Sample Sort, Histogram Sort, Exact-Splitting Sort, Radix Sort, and Mini Sort

	Sample	**Histogram**	**Exact-splitting**	**Radix**		**Mini**
				$k = 4$	$k = 8$	
Expectation	$\mathcal{O}(p^2 \log p)$	$\mathcal{O}(p)$	$\mathcal{O}(p \log^2 p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(\log^2 p)$
Lichtenberg						
Model	$\mathcal{O}(p^2 \log p)$	$\mathcal{O}(p)$	$\mathcal{O}(p^{1.75} \log p)$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(\log^2 p)$	$\mathcal{O}(p^{0.75} \log^2 p)$
$\bar{R}^2$	0.99	0.99	0.99	0.99	0.99	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{0.75}/\log p)$	$\mathcal{O}(\sqrt{p}/\log p)$	$\mathcal{O}(\log p)$	$\mathcal{O}(p^{0.75})$
Match	✓	✓	$\approx$	$\approx$	$\approx$	**x**
Juqueen						
Model	$\mathcal{O}(p^2 \log p)$	$\mathcal{O}(p^{0.75})$	$\mathcal{O}(p \log p)$	$\mathcal{O}(p^{1.25})$	$\mathcal{O}(\log p)$	$\mathcal{O}(\log^2 p)$
$\bar{R}^2$	0.99	0.99	0.99	0.99	0.94	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{-0.25})$	$\mathcal{O}(1)$	$\mathcal{O}(p^{1.25}/\log p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Match	✓	$\approx$	✓	**x**	✓	✓

match. Since both Exact-splitting sort and Mini sort are based on global median approximation, the reason for the non-matching models is similar. The difference between the Lichtenberg model and the Juqueen model in both cases is the same as well, that is $\mathcal{O}(p^{0.75})$.

6 Related Work

The scalability validation framework combines two earlier ideas, performance assertions [5] and automated empirical modeling of performance [6], into a new approach for practical, scalability-oriented software engineering. Performance assertions are source-code annotations that specify performance requirements in terms of conditional expressions consisting of performance metrics, program variables, and constants. At runtime, the expressions are instantiated with measurements and subsequently evaluated. After the execution finishes, violations are reported. If the number of processes is included in such an expression, performance assertions can be used to verify the compliance with scalability requirements as long as these can be specified in terms of performance data acquired during a single run. Even though assertions support tolerance thresholds, their design necessitates a rather precise notion of how the application should perform at a given number of processes. Because of the detailed understanding of the code and the underlying system this requires, it is often unrealistic to expect such a precise notion. Furthermore, it is rarely portable. The scalability validation framework, in contrast, requires users to specify scalability expectations in terms of the more prevalent asymptotic complexities, ignoring platform-dependent coefficients. Rather than looking at a single run, we determine and evaluate the growth rate of a given metric across multiple runs with an increasing number of processes. Thus, our approach would be more practical in the common scenario where the developer has only a vague idea of how the code scales.

The model generator we apply to create our performance models is based on the one used by Calotoiu et al. [6]. However, while their generator uses a manually configured search space, our extended generator builds the search space automatically around an expected performance model, leveraging the user's available knowledge. It means that it can also find exponential models—something which is not supported by the original generator. We also compute divergence models as an indicator of how the deviation would grow as the scale increases. We expect that our methodology integrates well with other performance modeling frameworks such as Palm [55] or PMaC [56]. Palm uses source code annotations to generate hierarchical performance models from formal descriptions. It provides a compiler that produces an instrumented executable that collects performance measurements and integrates them into the analytical models derived from the annotations. The PMaC framework, on the other hand, uses a modeling approach based on simulation to analyze the performance impact of hardware accelerators such as GPUs and FPGAs.

The main case study of scalability validation framework, namely the scalability of MPI implementations, was inspired by various MPI benchmarking efforts. Notably, *SKaMPI* [27] defined a way to accurately measure the runtime of collective operations, which was later extended in *NBCBench* [26], [57] and which we adapted for our work. Our idea of comparing the scalability of different parts of the target library was motivated by *mpicroscope* [10]. Instead of giving the users direct time metrics, the benchmark searches for violations of performance guidelines. One guideline, for example, states that `MPI_Allreduce` should take a smaller or equal amount of time when compared to `MPI_Reduce` followed by `MPI_Bcast`. A violation of this guideline suggests that there is some optimization flaw in an MPI implementation. Our approach, however, offers a different perspective since it evaluates the violations using empirical models of execution time. In this way, it takes into account much larger scales.

Algorithm engineering is an emerging field that combines algorithm theory with software engineering [7]. Growing complexities of both the algorithms and the hardware (e.g., parallelism, memory hierarchy, etc.) create a gap between promising algorithmic ideas and their practical use. Algorithm engineering aims to bridge this gap by adopting elements from software engineering. In other words, it defines a cycle that consists of four major phases, namely, design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses. Existing studies show examples of applying this cycle to a number of algorithmic problems [58]; in particular, to parallel external sorting [7].

7 Conclusion

In this study, we propose a new approach for engineering parallel applications and algorithms for extreme-scale systems. With our scheme, we identify scalability issues in libraries and algorithm implementations that are thought to be scalable and pinpoint possible performance bugs and room for improvement. In contrast to previous approaches, our technique only requires the performance engineer to have a vague (asymptotic) idea of the scalability.

To achieve this, our tool chain utilizes empirical performance modeling to generate models that describe the behavior of functions in a target HPC library. To understand the scaling behavior, users can model execution time as the number of processes increases, and divergence models, derived from the generated models, reveal how severe the discrepancy between the observed and expected performance is. We demonstrate the effectiveness of our mechanism using a number of use cases, namely MPI collective operations, the MAFIA code, and parallel sorting algorithms.

Our first case study, however, examines what is probably the most important library interface in HPC, that is, the MPI library. We chose to focus on it first because many commercially mature and well-tested implementations are available and clear performance expectations exist in the literature. We show how our approach enables MPI developers to spot scalability bugs early on, before commencing full-scale tests on the target supercomputer. For this, we used automated experiments on four different machines with five different MPI libraries, and our tool discovered a number of scalability issues that can be grouped into the following cases: (a) key collective functions on Juropa and Piz Daint display unexpected behavior; (b) the performance guideline $Allreduce \preceq Reduce + Bcast$ is potentially violated on Piz Daint; (c) memory consumption on Juropa limits the number of possible processes; (d) communicator duplication on Piz Daint consumes more memory than necessary; and (e) on Lichtenberg, Open MPI generally has a better scalability than Intel MPI.

The case studies, and parallel sorting in particular, show that our approach is beneficial for the algorithm engineering process in three aspects. First, the MAFIA case study demonstrates that we can use algorithmic parameters, such as cluster dimensionality k, to model sequential performance. Second, in the design phase of algorithm engineering, the framework can quickly provide models of execution time that can guide designers in the decision to reuse existing code or implement a new solution. Third, in the implementation phase of algorithm engineering, the framework can provide models that allow for a faster and easier way to compare alternative implementations. This alleviates the challenge algorithm engineers face of comparing several implementations of an algorithm across multiple architectures.

We conclude that our approach is a viable technique that can both point to limitations of the systems and provide important hints for improving the scalability of algorithms and libraries. We also expect that this will motivate the development of clear performance expectations for other parallel libraries, such as ScaLAPACK or the parallel BLAS.

Acknowledgments

This work was supported in part by the German Research Foundation (DFG) and the Swiss US National Science Foundation (SNSF) through the DFG Priority Programme 1648 *Software for Exascale Computing* (SPPEXA) in the project Catwalk and the DFG project ExtraPeak. Additional support was provided by the German Federal Ministry of Education and Research (BMBF) through the TaLPas project under grant no. 01IH16008D, and by the US Department of Energy through the PRIMA-X project under grant no. DE-SC0015524. Furthermore, we would like to thank Patrick Reisert for his contribution to the Intel MPI and Open MPI evaluation, as well as thank Andrew Adinetz and Marius Poke for their contribution to the MAFIA study. We also wish to express our gratitude to Elmar Peise and Christian Siebert for their parallel sorting code. We would also like to acknowledge the following organizations for providing us with access to their supercomputers: Jülich Supercomputing Centre (JSC), Swiss National Supercomputing Centre (CSCS), and the University Computing Center (Hochschulrechenzentrum) of Technische Universität Darmstadt. Finally, we would like to express our gratitude to the Cray software development team, and specifically Mark Pagel, for their help and support.

References

[1] J. Shalf, S. Dosanjh, and J. Morrison, “Exascale computing technology challenges,” in *Proc. 9th Int. Conf. High Perform. Comput. Comput. Sci.*, 2010, pp. 1–25.

[2] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, “Collective communication: Theory, practice, and experience,” *Concurrency Comput.: Practice Experience*, vol. 19, no. 13, pp. 1749–1783, Sep. 2007.

[3] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of collective communication operations in MPICH,” *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, 2005.

[4] C. Vömel, “ScaLAPACK’s MRRR algorithm,” *ACM Trans. Math. Softw.*, vol. 37, no. 1, 2010, Art. no. 1.

[5] J. S. Vetter and P. H. Worley, “Asserting performance expectations,” in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2002, pp. 1–13.

[6] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, “Using automated performance modeling to find scalability bugs in complex codes,” in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2013, pp. 45:1–45:12.

[7] P. Sanders, “Algorithm engineering – An attempt at a definition,” *Efficient Algorithms Lecture Notes Comput. Sci.*, vol. 5760, pp. 321–340, 2009.

[8] A. Adinetz, J. Kraus, J. Meinke, and D. Pleiter, “GPUMAFIA: Efficient subspace clustering with MAFIA on GPUs,” in *Proc. 19th Int. Eur. Conf. Parallel Distrib. Comput.*, 2013, pp. 838–849.

[9] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf, “Exascaling your library: Will your implementation meet your expectations?” in *Proc. 29th ACM Int. Conf. Supercomput.*, Jun. 2015, pp. 165–175.

[10] J. L. Träff, “mpicroscope: Towards an MPI benchmark tool for performance guideline verification,” in *Proc. Eur. MPI Users’ Group Meeting*, Sep. 2012, pp. 100–109.

[11] Extra-P – Automated performance-modeling tool, [Online]. Available: http://www.scalasca.org/software/extra-p, Accessed on: Apr. 24, 2018

[12] A. Vogel, et al., “10,000 performance models per minute - Scalability of the UG4 simulation framework,” in *Proc. 21st Int. Eur. Conf. Parallel Distrib. Comput.*, 2015, pp. 519–531.

[13] A. Calotoiu, et al., “Fast multi-parameter performance modeling,” in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2016, pp. 1–10.

[14] C. Iwainsky, et al., “How many threads will be too many? On the scalability of OpenMP implementations,” in *Proc. 21st Int. Eur. Conf. Parallel Distrib. Comput.*, 2015, pp. 451–463.

[15] S. Shudler, A. Calotoiu, T. Hoefler, and F. Wolf, "Isoefficiency in practice: Configuring and understanding the performance of task-based applications," in *Proc. 22nd ACM SIGPLAN Symp. Principles Practice Parallel Program.*, Feb. 2017, pp. 131–143.

[16] N. R. Draper and H. Smith, *Applied Regression Analysis*, 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2014.

[17] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "MPI on millions of cores," *Parallel Process. Lett.*, vol. 21, no. 1, pp. 45–60, Mar. 2011.

[18] D. Goodell, W. Gropp, X. Zhao, and R. Thakur, "Scalable memory use in MPI: A case study with MPICH2," in *Proc. Eur. MPI Users' Group Meeting*, Sep. 2011, pp. 140–149.

[19] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Program.*, vol. 17, no. 1, pp. 1–17, Feb. 1988.

[20] J. Bruck, C.-T. Ho, E. Upfal, S. Kipnis, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 11, pp. 1143–1156, Nov. 1997.

[21] MPICH – High-performance portable MPI, [Online]. Available: https://www.mpich.org, Accessed on: Apr. 24, 2018

[22] J. L. Träff, W. Gropp, and R. Thakur, "Self-consistent MPI performance guidelines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 5, pp. 698–709, May 2010.

[23] S. Hunold, A. Carpen-Amarie, F. D. Lübbe, and J. L. Träff, "Automatic verification of self-consistent MPI performance guidelines," in *Proc. 22nd Int. Eur. Conf. Parallel Distrib. Comput.*, 2016, pp. 433–446.

[24] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency Comput.: Practice Experience*, vol. 22, no. 6, pp. 702–719, Apr. 2010.

[25] A. Knüpfer, et al., "Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Proc. 5th Parallel Tools Workshop*, 2011, pp. 79–91.

[26] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately measuring collective operations at massive scale," in *Proc. 22nd IEEE Int. Parallel Distrib. Process. Symp.*, Apr. 2008, pp. 1–8.

[27] R. Reussner, P. Sanders, and J. L. Träff, "SKaMPI: A comprehensive benchmark for public benchmarking of MPI," *Sci. Program.*, vol. 10, no. 1, pp. 55–65, 2002.

[28] JuBE: Jülich benchmarking environment, [Online]. Available: http://www.fz-juelich.de/jsc/jube, Accessed on: Apr. 24, 2018

[29] JUQUEEN - Jülich blue Gene/Q, [Online]. Available: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html, Accessed on: Apr. 24, 2018

[30] Piz Daint supercomputer at Swiss National Supercomputing Centre (CSCS), [Online]. Available: http://www.cscs.ch/computers/piz_daint/index.html, Accessed on: Feb. 17, 2015

[31] S. Kumar, et al., "PAMI: A parallel active message interface for the Blue Gene/Q supercomputer," in *Proc. 26th IEEE Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 763–773.

[32] D. Chen, et al., "The IBM Blue Gene/Q interconnection network and message unit," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2011, pp. 26:1–26:10.

[33] ParaStation MPI user's guide, [Online]. Available: http://docs.par-tec.com/html/psmpi-userguide/index.html, Accessed on: Apr. 24, 2018

[34] T. Hoefler, T. Schneider, and A. Lumsdaine, "The impact of network noise at large-scale communication performance," in *Proc. 23rd IEEE Int. Parallel Distrib. Process. Symp.*, May 2009, pp. 1–8.

[35] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proc. 25th ACM Int. Conf. Supercomput.*, Jun. 2011, pp. 75–84.

[36] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2013, pp. 41:1–41:12.

[37] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Proc. IEEE Conf. Cluster Comput.*, Sep. 2006, pp. 1–12.

[38] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2010, pp. 1–11.

[39] P. Reisert, "Automated Refinement of Performance Models," Dept. of Computer Science, Master's thesis, Technische Universität Darmstadt, Darmstadt, Germany, Apr. 2017.

[40] Lichtenberg high performance computer of Technische Universität Darmstadt, [Online]. Available: http://www.hhlr.tu-darmstadt.de/hhlr/index.en.jsp, Accessed on: Apr. 24, 2018.

[41] S. Hunold and A. Carpen-Amarie, "MPI benchmarking revisited: Experimental design and reproducibility," *CoRR*, vol. abs/1505.07734, 2015, http://arxiv.org/abs/1505.07734

[42] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3617–3630, Dec. 2016.

[43] ReproMPI benchmark, [Online]. Available: https://github.com/hunsa/reprompi, Accessed on: Apr. 24, 2018

[44] K. Ilyas, A. Calotoiu, and F. Wolf, "Off-road performance modeling – How to deal with segmented data," in *Proc. 23rd Int. Eur. Conf. Parallel Distrib. Comput.*, Aug. 2017, pp. 36–48.

[45] M. Poke, "SymPtOM: Informed automatic performance modeling," Laboratory for Parallel Programming, Master's thesis, German Res. School Simulation Sci., Aachen, Germany, Oct. 2013.

[46] C. Siebert and F. Wolf, "A Scalable Parallel Sorting Algorithm Using Exact Splitting," RWTH Aachen Univ., Aachen, Germany, Tech. Rep. RWTH-CONV-008835, 2011.

[47] E. Solomonik and L. V. Kalé, "Highly scalable parallel sorting," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2010, pp. 1–12.

[48] M. Axtmann and P. Sanders, "Robust massively parallel sorting," in *Proc. 19th Workshop Algorithm Eng. Experiments*, 2017, pp. 83–97.

[49] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[50] G. E. Blelloch, et al., "A comparison of sorting algorithms for the connection machine CM-2," in *Proc. 3rd ACM Symp. Parallel Algorithms Archit.*, 1991, pp. 3–16.

[51] C. Siebert and F. Wolf, "Parallel sorting with minimal data," in *Proc. 18th Eur. MPI Users' Group Meeting*, Sep. 2011, pp. 170–177.

[52] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and S. Clifford, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: The MIT Press, 2009.

[53] Y. Berens, "Scalability validation of parallel sorting algorithms," Dept. of Computer Science, Bachelor's thesis, Technische Universität Darmstadt, Darmstadt, Germany, Oct. 2017.

[54] P. J. Rousseeuw and G. W. Bassett Jr, "The remedian: A robust averaging method for large data sets," *J. Amer. Statistical Assoc.*, vol. 85, no. 409, pp. 97–104, 1990.

[55] N. R. Tallent and A. Hoisie, "Palm: Easing the burden of analytical performance modeling," in *Proc. 28th ACM Int. Conf. Supercomput.*, Jun. 2014, pp. 221–230.

[56] M. R. Meswani, et al., "Modeling and predicting performance of high performance computing applications on hardware accelerators," *Int. J. High Perform. Comput. Appl.*, vol. 27, no. 2, pp. 89–108, May 2013.

[57] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2007, pp. 52:1–52:10.

[58] L. Kliemann and P. Sanders, *Algorithm Engineering: Selected Results and Surveys*. Berlin, Germany: Springer International Publishing, 2016.

Sergei Shudler received the BSc and MSc degrees in computer science from the Hebrew University of Jerusalem, and the PhD degree from Technische Universität Darmstadt, in 2018. He is a postdoctoral researcher at Argonne National Laboratory. In his work, he focuses on techniques to engineer parallel programs for extreme-scale systems. He also works on in-situ techniques to analyze and visualize data on-the-fly. Before starting the PhD program, he worked in the industry and co-developed a number of projects. His research interests include parallel programming, performance analysis, and machine learning.

Yannick Berens received the BSc degree in information systems technology from Technische Universität (TU) Darmstadt, in 2017. Currently, he is working toward the MSc degree and works at Continental Teves AG & Co. oHG. He worked as a student assistant with the Laboratory for Parallel Programming, TU Darmstadt. In his studies, he focused on software engineering and system programming. His research interests include concurrency and memory safety in software design along with parallel and functional programming.

Alexandru Calotoiu received the PhD degree from Technische Universität (TU) Darmstadt, in 2017, and was awarded the prize for best PhD thesis of the year in the computer science department for outstanding scientific performance by the Association of Friends of Technische Universität zu Darmstadt e.V. He works as a senior research associate with the group of Prof. Wolf. He has worked as part of the Laboratory for Parallel Programming at the German Research School for Simulation Sciences since 2011 and is now part of the Laboratory for Parallel Programming at the Computer Science Department of TU Darmstadt. His research interests are performance modeling, parallel programming, and machine learning. He is the main designer and developer of the automated performance-modeling tool Extra-P.

Torsten Hoefler is an associate professor of computer science at ETH Zürich, Switzerland. Before joining ETH, he led the performance modeling and simulation efforts of parallel petascale applications for the NSF-funded Blue Waters project at NCSA/UIUC. He is also a key member of the Message Passing Interface (MPI) Forum where he chairs the "Collective Operations and Topologies" working group. He won best paper awards at the ACM/IEEE Supercomputing Conference SC10, SC13, SC14, EuroMPI'13, HPDC'15, HPDC'16, IPDPS'15, and other conferences. He published numerous peer-reviewed scientific conference and journal articles and authored chapters of the MPI-2.2 and MPI-3.0 standards. He received the Latsis prize of ETH Zurich as well as an ERC starting grant in 2015. His research interests revolve around the central topic of "Performance-centric System Design" and include scalable networks, parallel programming techniques, and performance modeling. For additional information, please visit Torsten's homepage at htor.inf.ethz.ch.

Alexandre Strube received the MSc degree in advanced informatics and multimedia design, and the PhD degree in high-performance computing from the University Autònoma of Barcelona, in 2011. He is a scientific staff member with the Jülich Supercomputing Centre, where he works at the User Support and Application Optimization team, responsible for the optimization and scaling of applications in terms of performance, efficiency and parallel I/O. Between 2011 and 2016, he worked at the Performance Analysis group of the Jülich Supercomputing Centre, where he co-developed tools to analyze parallel programs. Between 2002 and 2006, he worked for the Ubuntu Linux project in partnership with a computer manufacturer, deploying hundreds of thousands of free software machines.

Felix Wolf received the PhD degree from RWTH Aachen University, in 2003. He is full professor with the Department of Computer Science of Technische Universität Darmstadt in Germany, where he leads the Laboratory for Parallel Programming. He works on methods, tools, and algorithms that support the development and deployment of parallel software systems in various stages of their life cycle. After working more than two years as a postdoc at the Innovative Computing Laboratory of the University of Tennessee, he was appointed research group leader at the Jülich Supercomputing Centre. Between 2009 and 2015, he was head of the Laboratory for Parallel Programming at the German Research School for Simulation Sciences in Aachen and full professor at RWTH Aachen University. He has published more than a hundred refereed articles on parallel computing, several of which have received awards.