Exascaling Your Library: Will Your Implementation Meet Your Expectations?

Sergei Shudler
TU Darmstadt
Darmstadt, Germany
shudler@cs.tu-darmstadt.de

Alexandru Calotoiu
TU Darmstadt
Darmstadt, Germany
calotoiu@cs.tu-darmstadt.de

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

Alexandre Strube
Jülich Supercomputing Centre
Jülich, Germany
a.strube@fz-juelich.de

Felix Wolf
TU Darmstadt
Darmstadt, Germany
wolf@cs.tu-darmstadt.de

ABSTRACT

Many libraries in the HPC field encapsulate 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 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 as an example, we show how our method can help uncover non-obvious limitations of both libraries and underlying platforms.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.4.8 [**Performance**]: Modeling and prediction; C.4 [**Performance of Systems**]: Modeling techniques; D.1.3 [**Concurrent Programming**]: Parallel programming

Keywords

software engineering; high performance computing; parallel programming; performance analysis

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.

http://dx.doi.org/10.1145/2751205.2751216.

1. INTRODUCTION

The most powerful supercomputers today allow computations to be run on millions of cores and in the not-so-distant future this number may grow to tens of millions and billions of cores. Since many applications critically depend on parallel libraries such as MPI, PETSc, ScaLAPACK and 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 encounter more challenges removing these bottlenecks in the libraries they are using.

Library developers, on the other hand, are confronted with the challenge 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 [7, 19, 22]. 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 automated performance modeling with performance expectations in a novel scalability test framework. Similar to performance assertions [21], 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 need 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

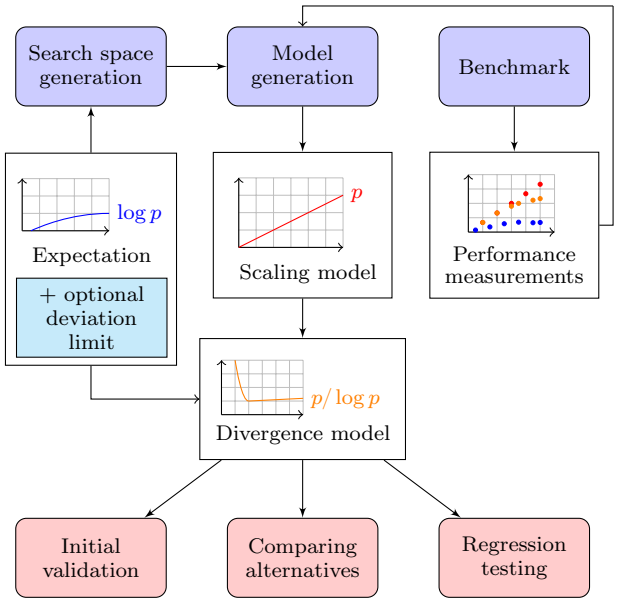

Figure 1: Framework overview including use cases.

behavior as they do, we tailor the model search spaces to expectations and also generate divergence models that help 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 later 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.

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:

- 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 second case study involving the MAFIA (Merging of Adaptive Finite IntervAls) code [3], we demonstrate that our approach is also applicable to algorithmic modeling. In this case, the model is a function of algorithm parameters.

The remainder of the paper is organized as follows: in Section 2, we provide an overview of our approach. Section 3 outlines how the framework must be instantiated for the MPI example and Section 4 then shows the experimental results. Since the MAFIA study relies largely on the same infrastructure, we chose to present it in the same section immediately after the MPI study. Finally, we review related work in Section 5, before drawing our conclusions in Section 6.

2. APPROACH

The objective of our approach, which is illustrated in Figure 1, 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*; (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 black-box hypothesis. A library call `sort(int *array, int N)`, for example, that sorts an integer array should perform the sorting in $\mathcal{O}(N \log N)$ rather than in $\mathcal{O}(N^2)$ steps.

In our expectation-centric performance modeling approach we assume that the user is a domain expert capable of providing initial expectations. However, before being able to define expectations, a user has to choose the library functions that will be subjected to the scaling analysis and the relevant scaling metrics. The more functions a 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. Very often, countable metrics such as floating-point operations yield better empirical scaling models because they are less prone to jitter. If only a hypothetical expectation is available, the model generator can use it to generate a model which 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. 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 3.2.

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. 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 growth of generated performance 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.

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 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. 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 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 the ticks this would create are too fine-grained to characterize significant deviations. Finally, we multiply each term in the search space with a coefficient placeholder that 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. However, if more flexibility is required, the user has the option of providing the deviation limit and modifying the search space. It means either refining the resolution by placing further exponents in gaps between existing terms or expanding the space beyond the default boundaries of B_l and B_u. However, there is a trade-off between accuracy and speed; therefore, applying these modifications will increase the model-generation time. As an approximate point of orientation, the entire modeling process in our case studies never took more than a few seconds per library.

As an example, let us consider the expectation $E(p) = p$. In this case, the default deviation limit is $D(p) = \sqrt{p}$ since it is exactly half of the power of p. The default lower and upper search space boundaries are respectively 1 and p^2. By averaging the exponents of adjacent terms in our search space we first construct the models $\sqrt{p}$ and $p\sqrt{p}$, and in the next step we add $\{p^j \mid j = \frac{1}{4}, \frac{3}{4}, \frac{5}{4}, \frac{7}{4}\}$. 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:

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

The model generator now needs a set of measurements as input whose precise nature depends on the scaling objective (e.g., number of processes vs. input size, weak vs. 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). It then starts searching the search space for the model that best fits the measurements and uses the adjusted coefficient of determination as an accuracy metric. The adjusted coefficient of determination is a standard statistical fit factor $\in [0, 1]$, with 1 indicating optimal fit.

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 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: 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. 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 [4]. This makes MPI an ideal case study for testing our approach. We now present the instantiation of each step in our test framework. 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 Expectations

We chose to focus on the most common MPI collective functions and latency-oriented runtimes (message sizes in the order of hundreds of bytes). Specifically, we looked at: *Barrier*, *Bcast*, *Reduce*, *Allreduce*, *Gather*, *Allgather*, and *Alltoall*. By focusing on latency, 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 also focus on the memory requirements of communicators, and measure the memory overhead of the *Comm_create*, *Comm_dup*, *Win_create*, and *Cart_create* functions. Lastly, we also chose to analyze the MPI memory consumption by estimating the process memory allocated during the benchmark execution.

The expectations for the runtime of collective operations in our MPI case study come from either the analysis of Chan et al. [7] or MPICH [19]. These particular cost models incorporate years of research and optimizations that make them a good reference for comparison. They are configurable, and can be changed as needed to reflect more specialized requirements. Table 2 presents these expectations for each collective operation. Many implementations of MPI collective functions (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 the analysis by Balaji et al. [4]. The memory overhead of communicator creation, either from a group or for a new Cartesian topology, is expected to be linear in its number of processes. Communicator duplication, on the other hand, requires only constant overhead and is therefore expected to remain constant as the number of processes grows. The creation of an RMA window (`MPI_Win_create`) is expected to be linear in the number of processes. In general, a scalable MPI library should consume a fixed amount of memory, independent of the number of processes [4]. Some libraries, however, require translation tables for ranks in MPI_COMM_WORLD to network ranks (e.g., IP addresses). However, this is suboptimal and should not consume more than a few bytes per MPI process in order to support highly scalable systems.

MPI performance guidelines specify internal performance consistency rules between MPI functions [20]. 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

execution time of `MPI_Allreduce` is not greater than the execution time of a combination of `MPI_Reduce` and `MPI_Bcast`. The same logic is also applied to the second guideline.

3.2 Benchmark Design

Although the benchmark we designed focuses on MPI, the general structure and principles can be adapted in other libraries as well. It consists of a series of smaller micro-benchmarks which 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. To obtain timings for collectives, we adopted the approach by Hoefler et al. [10], which first lets all processes start a collective operation at the same time, and then takes the maximum runtime across all processes. According to this method, we first calculate clock differences relative to the first process, and then set a time window relative to this process in which every process should start the operation.

A micro-benchmark starts with a number of warm-up runs and continues to execute the collective function R times. 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. To strike a balance between the number of repetitions and the total runtime of the benchmark, we empirically chose $R = 400$ for all the machines we used.

The memory allocated to a process on Linux and Unix-like systems is measured by analyzing the mapped memory regions in a */proc/self/smaps* file. 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 (CNK) 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 isolate the part which is used by MPI we first measure the allocated memory before MPI is initialized, and then subtract it from the measurement after all MPI functions have been executed and all user-created MPI data objects been freed, but before MPI is finalized. The additional memory for buffers and variables that were allocated by the micro-benchmarks is also subtracted from the estimate.

To help identify architectural constraints, or negative effects of neighbor network activity, we calibrate the benchmark by running a manually implemented binary-tree broadcast [7] 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, e.g., 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) [1], 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 then iterates over these ranges independently and creates a batch job for each combination.

Table 1: Machine specifications (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

3.3 Generation of Scaling Models

The inputs of the model generation phase are runtimes of collective functions, communicator memory overheads, and the estimate of the memory allocated by MPI, 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. In any case, the divergence model in the average case is as big as in the best case.

As depicted in Tables 2 and 3, there were four different expectations in this case study: $\mathcal{O}(1)$, $\mathcal{O}(\log p)$, $\mathcal{O}(p)$, and $\mathcal{O}(p \log p)$. 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 $F_1(x)$, thus also expanding the default search space boundaries. For more consistency, we decided to refine the largest default search space we had, which was the 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.

3.4 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 2 is an example of such a list. The divergence model and the match indicator were already discussed in Section 2.4.

4. EVALUATION

In this section, we analyze the results of our experiments. We used three different machines and MPI implementations, and, as already explained, measured the runtime of collec-

tive functions, the memory overhead of communicators, and the memory allocated by the process during the benchmark execution. We first present the machines and the experimental setup and then discuss the results.

4.1 Experimental Setup

Table 1 presents the specifications of the three machines on which we conducted our experiments and tested our approach. Juqueen is a Blue Gene/Q machine built by IBM and it is the capability supercomputer at Forschungszentrum Jülich (FZJ). It is specifically designed for highly scalable codes and features improved energy efficiency. The specialized CNK kernel on the compute nodes reduces jitter and allows for reproducible measurements. Juropa, on the other hand, is the capacity machine at FZJ and is based on an Intel architecture. Piz Daint, an x86-based Cray-XC30 machine at the Swiss National Supercomputing Centre (CSCS), was built by Cray, and therefore has both a different network topology and a different MPI implementation. 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 [13] and uses special hardware components to accelerate collective functions [8]. Users have a choice of various protocols for some of the frequently used collective functions, e.g., binary-tree or binomial for `MPI_Allreduce`. They also have the option to revert to the plain MPICH implementation from which the Blue Gene version was derived. For some numbers of processes and message sizes, the special hardware components have no tangible benefits; in these cases, the implementation might revert automatically to the original MPICH algorithm. Juqueen provides an extension of MPI that makes it possible to query which algorithm was actually used during the execution of the collective function. 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 study on supported implementations, that is, PAMI on Juqueen, ParaStation MPI on Juropa, and CrayMPI on Piz Daint. We chose to set the number of MPI processes per node to be the same as 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 and means that the application would need to have more threads.

All three machines in our experiments provide highly accurate, high-resolution hardware timers in the form of registers that can be read very quickly with an atomic instruction: MFTB on PowerPC, and RDTSC on x86. They allow individual runtimes to be measured instead of executing a function N times in a loop, measuring the total runtime, and then dividing it by N to get an average. The latter approach suffers from pipeline effects and tends to underestimate the latency [10]. As mentioned before, each run of the benchmark performs R iterations. In order to capture performance fluctuations due to topology and network noise, we repeated the runs 10 times. Each run was submitted separately to the batch system and thus was given a different node allocation. We note that the experiments on Piz Daint were performed with default Cray MPI library optimizations. The newer version of Cray MPI has additional algorithms that may improve scaling, and can be used by setting appropriate environment variables.

Table 2: Generated (empirical) runtime models of collective functions on Juqueen, Juropa, and Piz Daint alongside their theoretical expectations.

	Juqueen	Juropa	Piz Daint
Barrier			Expectation: $\mathcal{O}(\log p)$
Model	$\mathcal{O}(\log p)$	$\mathcal{O}(p^{0.67}\log p)$	$\mathcal{O}(p^{0.33})$
$\bar{R}^2$	0.99	0.99	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{0.67})$	$\mathcal{O}(p^{0.33}/\log p)$
Match	✓	x	≈
Bcast			Expectation: $\mathcal{O}(\log p)$
Model	$\mathcal{O}(\log p)$	$\mathcal{O}(p^{0.5})$	$\mathcal{O}(p^{0.5})$
$\bar{R}^2$	0.86	0.98	0.94
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{0.5}/\log p)$	$\mathcal{O}(p^{0.5}/\log p)$
Match	✓	≈	≈
Reduce			Expectation: $\mathcal{O}(\log p)$
Model	$\mathcal{O}(\log p)$	$\mathcal{O}(p^{0.5}\log p)$	$\mathcal{O}(p^{0.5}\log p)$
$\bar{R}^2$	0.93	0.99	0.94
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{0.5})$	$\mathcal{O}(p^{0.5})$
Match	✓	≈	≈
Allreduce			Expectation: $\mathcal{O}(\log p)$
Model	$\mathcal{O}(\log p)$	$\mathcal{O}(p^{0.5})$	$\mathcal{O}(p^{0.67}\log p)$
$\bar{R}^2$	0.87	0.99	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{0.5}/\log p)$	$\mathcal{O}(p^{0.67})$
Match	✓	≈	x ⚠
Gather			Expectation: $\mathcal{O}(p)$
Model	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p)$
$\bar{R}^2$	0.99	0.99	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Match	✓	✓	✓
Allgather			Expectation: $\mathcal{O}(p)$
Model	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p^{1.25})$
$\bar{R}^2$	0.99	0.98	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{0.25})$
Match	✓	✓	≈
Alltoall			Expectation: $\mathcal{O}(p\log p)$
Model	$\mathcal{O}(p)$	$\mathcal{O}(p^{1.25})$	$\mathcal{O}(p^{1.33})$
$\bar{R}^2$	0.99	0.99	0.99
$\delta(p)$	$\mathcal{O}(1/\log p)$	$\mathcal{O}(p^{0.25}/\log p)$	$\mathcal{O}(p^{0.33}/\log p)$
Match	≈	≈	≈
Bcast (BT)			Expectation: $\mathcal{O}(\log p)$
Model	$\mathcal{O}(\log p)$	$\mathcal{O}(p^{1.25}\log p)$	$\mathcal{O}(p\log p)$
$\bar{R}^2$	0.99	0.99	0.99
$\delta(p)$	$\mathcal{O}(1)$	$\mathcal{O}(p^{1.25})$	$\mathcal{O}(p)$
Match	✓	x	x

4.2 Result Analysis

In this subsection, we present the results of our analysis. Tables 2 and 3 present the models as formulae next to our expectations. Table 2 refers to runtime and Table 3 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. Then follows the δ row with the divergence models 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, a ≈ is shown. A solid × represents an unquestionable mismatch. A warning sign ⚠ indicates the violation of a performance guideline. Figure 2 presents the runtime models for the collective functions we benchmarked plus the estimation of memory consumption as graphs. The circles, squares, and triangles depict the actual measurements, whereas the lines are the predictions. Each curve is annotated with the corresponding model which 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 below starts with Juqueen, on which almost all the generated models correspond almost fully to expectations. We then continue to 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 2, 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 binary-tree (BT) version of the broadcast is shown at the bottom of Table 2. The expected cost of this algorithm for small messages is: $(\alpha + \beta) \log p$; and though it is slower in absolute terms than the native `MPI_Bcast`, the generated model is still logarithmic. It serves as a strong indicator that other factors have minimal influence on the runtime. Table 3 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 did 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`, did not correspond to the expected logarithmic models. It is important to note that `MPI_Barrier` can have both logarithmic and linear implementations [15]. However, since the optimized MPICH implementation is logarithmic [19], we expect it to behave logarithmically on Juropa as well. The generated model of the binary-tree (BT) broadcast falls outside the deviation limits and clearly fails to match the expected logarithmic model. 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:

Table 3: Generated (empirical) runtime 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	✓	×	✓
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	✓	✓	×
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	✓	✓	✓

1. The network model which was used to calculate the expected cost of the binary-tree broadcast algorithm is simpler than the IB fat-tree interconnect on Juropa.

2. On some machines, the performance of communication-heavy applications strongly depends on the node allocation they receive and the neighborhood of each node [5]. An application which runs on a neighbor node and produces heavy network load creates more perturbation for our benchmark.

3. Network hardware and topology can influence the runtime of various collective functions and make them slower than expected [11, 12].

The above factors can also offer an explanation for the discrepancies between the generated models and the expectations on Piz Daint. 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 2, the warning sign under *Match* signals that a performance guideline violation was detected. As discussed in Section 3.4, 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

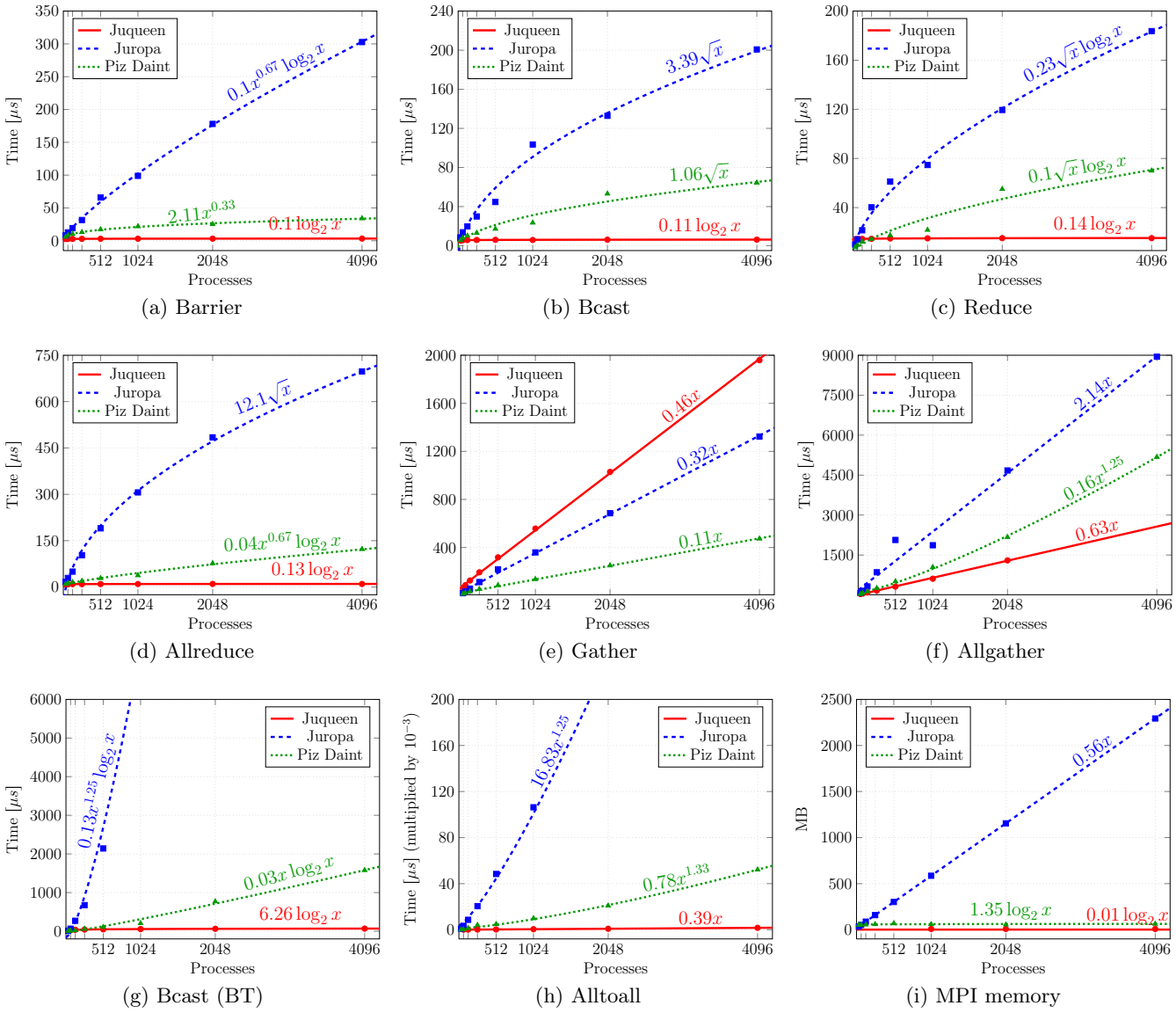

Figure 2: Measurements (circles, squares, triangles) and generated runtime models (plot lines) on Juqueen, Juropa, and Piz Daint.

that the guideline would be violated at larger scales. However, whether this is indeed the case must be verified in future experiments.

The communicator memory overheads on Juropa and Piz Daint are presented in Table 3. 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` is taken from MPICH 3.1.2, and that the MPICH version behaves in the same manner. This result clearly shows that there might be a scalability bug in this function; further study is required to find ways to fix it.

Figure 2i presents the models for the estimated fraction of the process memory that is consumed by MPI 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 [2]. When using `MPI_Alltoall` each process will allocate $0.55p$ MB of memory, which is exactly the linear behavior we discovered through our experiments.

Table 4: Generated (empirical) runtime models of MAFIA functions alongside their theoretical expectations.

	gen	**dedup**	**pcount**	**unjoin**
Expectation	$\mathcal{O}(k^3 2^k)$	$\mathcal{O}(k^4 2^k)$	$\mathcal{O}(k2^k)$	$\mathcal{O}(k^3 2^k)$
Model	$\mathcal{O}(k^4 2^k)$	$\mathcal{O}(k^4 2^k)$	$\mathcal{O}(k2^k)$	$\mathcal{O}(k^2 2^k)$
$\delta(k)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1/k)$
Match	$\approx$	✓	✓	$\approx$

4.3 MAFIA

No matter how large the degree of parallelism, optimizing sequential code is still essential to achieve good performance. The subject of our second case is therefore MAFIA (Merging of Adaptive Finite IntervAls), a sequential data-mining program utilizing a collection of key routines. 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 [3]. 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 our 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, *gen*, *dedup*, *pcount*, and *unjoin*. They correspond to the generation of candidate sets, de-duplication of them, identifying dense sets, and checking whether lower dimensional sets were not already absorbed by the higher ones [16] respectively. Table 4 presents the expectations for these functions provided by Adinetz et al. [3] in their effort to optimize MAFIA.

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. 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 experiments were conducted on one node of Juropa and repeated for $k = 3, 4, ..., 16$. In contrast to the MPI study, all the expectations were exponential: $\mathcal{O}(k2^k)$, $\mathcal{O}(k^3 2^k)$, and $\mathcal{O}(k^4 2^k)$. In all of these cases we did not change the default deviation limits or the search space boundaries. As Table 4 shows, all the generated models matched our expectations completely or were inside the deviation limits.

This example illustrates the flexibility of our approach, which can be adapted to different scalability problems with different expectations.

5. RELATED WORK

Our approach combines two earlier ideas, performance assertions [21] and automated empirical performance modeling [6], into a new approach for practical performance-centric code design. 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. 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/or the underlying system this requires, it is often unrealistic to expect such a precise notion. Furthermore, it is rarely portable. Our approach, in contrast, specifies 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 similar in spirit to the one used by Calotoiu et al. [6]. 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 their generator. Another difference is that we 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 [18] or the PMaC tools [14].

Our case study, the scalability analysis of MPI implementations, was inspired by various MPI benchmarking efforts. Notably, *SKaMPI* [17] defined a way to accurately measure collective operations [23], which was later extended in *NBCBench* [9, 10] and which we adapted for our work. Our idea of comparing the scalability of different parts of the target library was motivated by *mpicroscope* [20]. 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. However, our approach offers a different perspective since it automatically evaluates the guidelines using the generated models, and thus can predict violations for a large number of processes.

6. CONCLUSION

In this paper, we propose a new software engineering discipline for extreme-scale systems. With our scheme, we identify scalability issues in libraries which 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, although the accuracy improves if more information is available (e.g., a performance litmus test or more precise expectations). We also supply a tool chain that automates large parts of our four-step process and is ready for immediate use by performance engineers.

To achieve this, our tool chain utilizes automated performance modeling to generate analytical models of the runtime and memory overheads of selected library functions. Divergence models derived from the generated models show how the actual behavior differs when increasing the number of processes, revealing potential performance problems in the implementation.

We demonstrate the effectiveness of our mechanism using what is probably the most important library interface in HPC: the Message Passing Interface. We chose it as a use case 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 three different machines with three 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. We conclude that our approach is a viable technique that can both point to limitations of the systems and provide MPI developers with important hints for improving the scalability of their implementations. We also expect that this will motivate the development of clear performance expectations for other parallel libraries such as ScaLAPACK or the parallel BLAS.

7. ACKNOWLEDGMENTS

This work was supported in part by the German Research Foundation (DFG) and the Swiss National Science Foundation (SNSF) through the DFG Priority Programme 1648 *Software for Exascale Computing* (SPPEXA). We would like to thank Andrew Adinetz and Marius Poke for their contribution to the MAFIA study. We would also like to acknowledge Jülich Supercomputing Centre (JSC) and Swiss National Supercomputing Centre (CSCS), which gave us access to their supercomputers. Finally, we would like to express our gratitude to the Cray software development team, and specifically Mark Pagel, for their help and support.

8. REFERENCES

[1] JuBE: Jülich Benchmarking Environment. `http://www.fz-juelich.de/jsc/jube`.

[2] ParaStation MPI User's Guide. `http://docs.par-tec.com/html/psmpi-userguide/index.html`.

[3] A. Adinetz, J. Kraus, J. Meinke, and D. Pleiter. GPUMAFIA: Efficient Subspace Clustering with MAFIA on GPUs. In *Proc of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 838–849. Springer-Verlag, 2013.

[4] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Traeff. MPI on Millions of Cores. *Parallel Processing Letters (PPL)*, 21(1):45–60, Mar. 2011.

[5] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs. In *Proc. of the ACM/IEEE Conference on Supercomputing*, SC '13, pages 41:1–41:12. ACM, 2013.

[6] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proc. of the ACM/IEEE Conference on Supercomputing*, SC '13, pages 45:1–45:12. ACM, 2013.

[7] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective Communication: Theory, Practice, and Experience. *Concurr. Comput. : Pract. Exper.*, 19(13):1749–1783, Sep 2007.

[8] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q Interconnection Network and Message Unit. In *Proc. of the ACM/IEEE Conference on Supercomputing*, SC '11, pages 26:1–26:10. ACM, 2011.

[9] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proc. of the ACM/IEEE Conference on Supercomputing*, SC '07, pages 52:1–52:10. IEEE Computer Society/ACM, 2007.

[10] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately Measuring Collective Operations at Massive Scale. In *Proc. of the IEEE International Parallel & Distributed Processing Symp.*, IPDPS '08, pages 1–8, 2008.

[11] T. Hoefler, T. Schneider, and A. Lumsdaine. The Impact of Network Noise at Large-Scale Communication Performance. In *Proc. of the IEEE International Parallel & Distributed Processing Symp.*, IPDPS '09, pages 1–8, 2009.

[12] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proc. of the ACM International Conference on Supercomputing*, ICS '11, pages 75–84. ACM, 2011.

[13] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *Proc. of the IEEE International Parallel & Distributed Processing Symp.*, IPDPS '12, pages 763–773. IEEE Computer Society, 2012.

[14] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. Modeling and Predicting Performance of High Performance Computing Applications on Hardware Accelerators. *Int. J. High Perform. Comput. Appl.*, 27(2):89–108, May 2013.

[15] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance

Analysis of MPI Collective Operations. *Cluster Computing*, 10(2):127–143, 2007.

[16] M. Poke. SymPtOM: Informed Automatic Performance Modeling. Master's thesis, German Research School for Simulation Sciences, Aachen, Germany, Oct 2013.

[17] R. Reussner, P. Sanders, and J. L. Träff. SKaMPI: a comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[18] N. R. Tallent and A. Hoisie. Palm: Easing the Burden of Analytical Performance Modeling. In *Proc. of the ACM International Conference on Supercomputing*, ICS '14, pages 221–230. ACM, 2014.

[19] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[20] J. L. Träff. mpicroscope: Towards an MPI Benchmark Tool for Performance Guideline Verification. In *Proc. of the European MPI Users' Group Meeting*, EuroMPI '12, pages 100–109. Springer-Verlag, 2012.

[21] J. S. Vetter and P. H. Worley. Asserting Performance Expectations. In *Proc. of the ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–13. IEEE Computer Society Press, 2002.

[22] C. Vömel. ScaLAPACK's MRRR algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 37(1), 2010.

[23] T. Worsch, R. Reussner, and W. Augustin. On Benchmarking Collective MPI Operations. In *Proc. of the European PVM/MPI Users' Group Meeting*, pages 271–279. Springer-Verlag, 2002.