Automatic Empirical Performance Modeling of Parallel Programs

Automatisches empirisches Leistungsmodellieren paralleler Programme
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von M.Sc. Dipl-Ing. Alexandru Calotoiu aus Bukarest, Romania
Tag der Einreichung: 15.9.2017, Tag der Prüfung: 6.10.2017
Darmstadt — 2018 — D 17

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Prof. Dr. Torsten Hoefler

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Laboratory for Parallel Programming

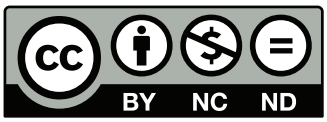
Automatic Empirical Performance Modeling of Parallel Programs
Automatisches empirisches Leistungsmodellieren paralleler Programme

Genehmigte Dissertation von M.Sc. Dipl-Ing. Alexandru Calotoiu aus Bukarest, Romania

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Prof. Dr. Torsten Hoefler

Tag der Einreichung: 15.9.2017
Tag der Prüfung: 6.10.2017

Darmstadt — 2018 — D 17



To
my wife Stanca Iulia

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 15.9.2017

_____

(Alexandru Calotoiu)

Abstract

Many parallel applications suffer from latent performance limitations that may prevent them from scaling to larger machine sizes or solving larger problems. Often, such performance bugs manifest themselves only when the code is put into production, a point where remediation can be difficult. Manually creating analytical performance models provides insights into optimization opportunities but is extremely costly if done for applications of realistic size. The effort limits application developers to only attempt it at most for a few selected kernels, running the risk of missing harmful bottlenecks. Furthermore, tuning large applications requires a clever exploration of the design and configuration space. Especially on supercomputers, this space is so large that its exhaustive traversal via performance experiments becomes too expensive, if not impossible. If we have to consider multiple performance-relevant parameters and their possible interactions at the same time, a common requirement in many situations, this task becomes even more complex.

The initial contribution of this thesis is a method to substantially improve both coverage and speed of performance modeling and analysis. Generating an empirical performance model automatically for each part of a parallel program with respect to the variation of a relevant parameter such as process count or problem size, it becomes possible to easily identify those parts that will reduce performance at larger core counts or when solving a bigger problem.

In the next step, we extended the approach with a method capable of modeling any combination of multiple execution parameters simultaneously, provided sufficient performance measurements are available. Multi-parameter modeling has so far been outside the reach of automatic methods due to the exponential growth of the model search space. Specialized heuristics developed as part of this work traverse the search space rapidly and generate insightful performance models that enable a wide range of uses from performance predictions for balanced machine design to performance tuning.

Finally we present a method that employs automated performance modeling to quickly predict application requirements for varying scales and problem sizes. Following this approach, it is possible to determine future requirements of major scientific applications, derive an optimization strategy, and illustrate system design tradeoffs in the light of their requirements. This supports the co-design process by informing hardware acquisition decisions with the actual needs of the software.

The methods described in this work are implemented in the performance analysis tool Extra-P. Extra-P has been released as open source and has been successfully used to gain insight into the performance of numerous scientific applications from a large range of fields. Since its release, Extra-P has an impact on the HPC community. Developers at both universities and research centers have used Extra-P to better understand the performance of their research codes.

Tutorials on the use of Extra-P have been offered at international conferences such as EuroMPI and Supercomputing further demonstrating the effectiveness of this approach in making performance modeling available to developers without requiring expert knowledge of the topic.

This work simplifies and streamlines the performance modeling process, offering insights into application behavior quickly and automatically and allowing the developer to focus on transforming these insights into tangible performance improvements.

Zusammenfassung

Viele parallele Anwendungen leiden unter latenten Leistungsbeschränkungen, die sie daran hindern können, auf größere Maschinen zu skalieren oder größere Probleme zu lösen. Oft manifestieren sich solche Leistungsfehler nur, wenn der Code in Produktion verwendet wird, ein Punkt, wo die Korrektur schwierig sein kann. Manuelles Erstellen von analytischen Leistungsmodellen bietet Einblicke in Optimierungsmöglichkeiten, ist aber äußerst kostspielig, wenn es für Anwendungen realistischer Größe angewandt wird. Die Bemühungen beschränken Anwendungsentwickler es nur für ein paar ausgewählte Programmteile zu versuchen, wobei das Risiko besteht, schädliche Engpässe zu übersehen. Darüber hinaus erfordert das Tuning großer Anwendungen eine geschickte Erforschung des Design- und Konfigurationsraums. Vor allem auf Supercomputern ist dieser Raum so groß, dass seine umfangreiche Durchquerung über Performance-Experimente zu teuer wird, wenn nicht unmöglich. Wenn wir in vielen Situationen mehrere leistungsrelevante Parameter und deren mögliche Wechselwirkungen gleichzeitig berücksichtigen müssen, wird diese Aufgabe noch komplexer.

Der erste Beitrag dieser Arbeit ist eine Methode, um sowohl die Abdeckung als auch die Geschwindigkeit der Leistungsmodellierung und -analyse wesentlich zu verbessern. Durch die Generierung eines empirischen Leistungsmodells für jeden Teil eines parallelen Programms in Bezug auf die Variation eines relevanten Parameters wie Prozesszählung oder Problemgröße wird es möglich, die Teile leicht zu identifizieren, die die Leistung bei größeren Kernzahlen oder bei der Lösung von größeren Problemen beschränken.

Im nächsten Schritt haben wir den Ansatz mit einer Methode erweitert, die in der Lage ist, jede Kombination mehrerer Ausführungsparameter gleichzeitig zu modellieren sofern genügend Leistungsmessungen vorliegen. Die Multi-Parameter-Modellierung ist bisher aufgrund des exponentiellen Wachstums des Modellsuchraums außerhalb der Reichweite von automatischen Methoden. Spezielle Heuristiken, die als Teil dieser Arbeit entwickelt wurden, durchqueren den Suchraum schnell und erzeugen aufschlussreiche Leistungsmodelle, die eine breite Palette von Einsatzmöglichkeiten von Leistungsvorhersagen für ausgewogenes Maschinendesign bis hin zur Performance-Optimierung ermöglichen.

Schließlich stellen wir eine Methode vor, die eine automatisierte Leistungsmodellierung einsetzt, um die Anwendungsanforderungen für unterschiedliche Skalen und Problemgrößen schnell vorherzusagen. Nach diesem Ansatz ist es möglich, zukünftige Anforderungen von großen wissenschaftlichen Anwendungen zu ermitteln, eine Optimierungsstrategie abzuleiten und Systemdesign-Kompromisse im Lichte ihrer Anforderungen zu veranschaulichen. Dies unterstützt den Co-Design-Prozess durch das Informieren von Hardware-Akquisitionsentscheidungen mit den tatsächlichen Bedürfnissen der Software.

Die in dieser Arbeit beschriebenen Methoden wurden im Leistungsanalyse-Tool Extra-P implementiert. Extra-P wurde als Open Source freigegeben und wurde erfolgreich eingesetzt, um Einblicke in die Leistungsfähigkeit zahlreicher wissenschaftlicher Anwendungen aus einer Vielzahl von Bereichen zu gewinnen. Seit seiner Veröffentlichung hat Extra-P einen Einfluss auf die HPC-Community. Entwickler an sowohl Universitäten und Forschungszentren haben Extra-P verwendet, um die Leistung ihrer Forschungscodes besser zu verstehen.

Tutorials über die Verwendung von Extra-P wurden auf internationalen Konferenzen wie EuroMPI und Supercomputing angeboten, die die Wirksamkeit dieses Ansatzes für Entwickler auch ohne Annahme von Fachwissen weiter demonstrieren.

Diese Arbeit vereinfacht und optimiert den Performance-Modellierungsprozess und bietet schnell und automatisch Einblicke in das Anwendungsverhalten und ermöglicht es dem Entwickler, sich auf die Umwandlung dieser Erkenntnisse in greifbare Leistungsverbesserungen zu konzentrieren.

Acknowledgements

While working for my PhD project, I have learned more than I had in all the previous years of study. I had the opportunity to cooperate with wonderful people on a topic that interests me greatly, and I had the freedom to explore promising research avenues. I am deeply grateful to my advisor, Prof. Dr. Felix Wolf, for his support, guidance and feedback at every step of my project. I am also grateful to my co-advisor, Prof. Dr. Torsten Hoefler, for the close collaboration and guidance during the entire duration of my doctoral studies.

I thank my colleagues at the German Research School for Simulation Sciences and at the Technical University Darmstadt for the work environment they provided and the support they offered. I especially want to thank Marc-André Hermanns and Marius Poke for the many helpful, insightful discussions as well as the constructive criticism they provided as I was defining the direction of my research, and Petra Stegmann for the feedback she provided on my doctoral thesis.

Furthermore, I thank Dr. Martin Schulz for the chance to experience working in the exceptional community at the Lawrence Livermore National Laboratory. I also acknowledge Dr. Bernd Mohr for hosting me at the Jülich Supercomputing Centre during the transition period from Aachen to Darmstadt and for providing me with computing resources whenever very large experiments where necessary in my research.

Finally, I am deeply grateful to my family and their never-ending patience during busy or stressful times and their unconditional and unwavering support throughout the project.

Contents

List of Figures

List of Tables

1 Introduction

This chapter provides the motivation for this thesis, presents its objectives, and outlines its contributions. This chapter starts by sketching the limitations of performance analysis that the method presented in this thesis eliminates and which define the objectives of this method. The realization of the objectives through the main contributions is then briefly described.

1.1 High-Performance Computing

Computers have revolutionized the world in the last decades. From communication and entertainment to industry and finance there is no field which was not revolutionized by computers and their explosive development. At the forefront of this tidal wave of innovation, science and engineering massively profit from these developments. Computers can solve problems which are too complex to manually attempt and perform virtual experiments which are too expensive or even too dangerous to carry out in reality. Simulations have joined theory and experiment as a third pillar supporting research and engineering.

The drive towards performance and efficiency means that designs for planes, cars, engines are first simulated on computers before a prototype is even built. The myriad applications of computation fluid dynamics, or CFD, are proof of the usefulness of computer-aided design. From pharmaceutical drug design to theoretical solid state physics, many fields of science use computer simulations as a valuable tool in the research process.

The quality and therefore usefulness of simulations depends on their level of detail. The more detail a simulation takes into account, the more resources it requires. There are, and probably always will be, questions that require computational speed or memory which a single desktop machine cannot provide. Using parallel processing techniques and vast parallel architectures, High-Performance Computing (HPC) is a research tool used to answer such questions.

The parallel nature of HPC applications combined with the complex distributed hardware systems make achieving optimal performance a very difficult challenge. Making sure that the available resources are used in a purposeful way is all the more important when considering the energy costs of running supercomputers. The most powerful supercomputers today use between 10 and 20 Megawatts, which means they cost over 10 million € per year just in power bills. Therefore, optimizing HPC applications is paramount to minimizing running costs.

Moore's Law observes that the number of transistors in an integrated circuit doubles roughly every eighteen months. However, due to physical limitations in the last years keeping up with Moore's Law meant that parallelism was required. From multi-core architectures to many-core systems, even mobile and embedded devices turn to parallelism to provide increased computational power. With parallelism on the rise and multi- and manycore systems becoming the norm in commodity and even personal computing, the lessons learned in HPC are necessary to the wider field of computer science to ensure adequate resource utilization of modern architectures.

At the same time, ever-growing application complexity across all domains, including but not limited to theoretical physics, fluid dynamics, or climate research requires a continuous focus on performance to productively use the large-scale machines that are being procured. However, designing such large applications is a complex task demanding foresight since they require large time investments in development as well as verification and are therefore meant to be used for decades. Thus, it is important that applications are efficient and potential bottlenecks are identified early in their design as well as throughout their whole life cycle.

1.2 Scalability

A core issue of parallel computing is developing algorithms and applications which scale well as the parallel architectures they run on evolve. This means algorithms must efficiently and meaningfully utilize the resources of these systems to solve ever larger problems, solve problems faster, or even both at the same time. Two types of scalability are often encountered in HPC: strong and weak scaling.

Strong scaling is defined as solving a given problem faster by increasing the number of processes assigned to it. Amdahl's law defines the maximum theoretical speedup of a program when using strong scaling, by linking this maximum to the degree to which a program can be parallelized as shown in Equation 1.1. The efficiency of strong scaling is limited: the more processes are used, the faster the actual problem is solved, but any overheads will take an increasingly larger percentage of the total run-time.

$$speedup = \frac{1}{fraction_{serial} + \frac{fraction_{parallel}}{processes}} \tag{1.1}$$

Weak scaling is defined as solving a larger problem with an increasing number of processes, such that the problem assigned to each process stays constant. This allows much larger process numbers to be used efficiently as it is possible to improve efficiency by sufficiently increasing the problem to be solved. While the concept of weak scaling is intuitive, making sure that the problem assigned to each process stays constant is far from a trivial task: for any but the simplest applications the amount of computation generated by an increasing problem size is not necessarily in a linear relation to it. Decomposing a problem into smaller problems to be handled by individual processes can also be difficult, and often incurs overheads. This is by no means the only difficulty in scaling applications.

When considering both weak and strong scaling of codes to larger numbers of processors, many HPC application developers face the situation that all of a sudden a part of the program starts consuming an excessive amount of time. Unfortunately, discovering latent scalability bottlenecks through experience is painful and expensive. Removing them requires not only potentially numerous large-scale experiments to track them down, prolonged by the scalability issue at hand, but often also major code surgery in the aftermath. All too often, this happens at a moment when the manpower is needed elsewhere. This is especially true for applications on the path to exascale, which have to address numerous technical challenges simultaneously, ranging from heterogeneous computing to resilience. Since such problems usually emerge at a

Figure 1.1: Performance projection overview [1].

later stage of the development process, dependencies between their source and the rest of the code that have grown over time can make remediation even harder.

1.3 Performance analysis

Improving the execution time of an application or making sure available resources are used efficiently is a very difficult task. Performance analysis is a wide term encompassing all methods used to better understand the behavior of applications either theoretically or when executed on particular systems. The goal of such an analysis is to identify any performance limitations that either the application or the hardware system have. By focusing on these limitations, the effort of hardware and software developers can be directed to where it will have the biggest impact. For example, if a program has one process that creates and assigns tasks to other processes and it cannot assign tasks as fast as the other processes solve them, improving the speed at which tasks are solved, will do nothing to improve overall performance. Instead, the task creation and assignment process should be parallelized.

Depending on how the performance analysis is performed we can classify the different approaches into experiment-based, simulations and analytical modeling. They each have advantages and disadvantages due to the effects the number of parameters considered by the analysis has on the results. The more parameters are considered, the more accurate is the result. However, the number of scenarios the result is applicable to is reduced. When fewer parameters are considered, the error of the analysis often grows. This trade-off is displayed in Figure 1.1 and is discussed in more detail by Hoefler et al. [1].

1.3.1 Experiments

A classic approach to performance analysis is for developers to simply run their application on the target system — effectively using the code as a benchmark of its own performance or as a measure of how well the application runs on a particular hardware system. Developers would select one or multiple configurations considered representative for how their application would be used in practice.

In this approach, application developers measure different performance metrics such as runtime and then focus on improving the most resource-intensive parts of their codes. This approach is often limited to discovering the low-hanging fruits with respect to optimization potential and usually provides no guarantees as to how close the performance achieved is to

the optimum on the chosen platform. When using benchmarking results to guide optimization, developers often exhaust their budget and capacity without a clear performance expectation.

While providing the most accurate results, benchmarking also makes interpreting results difficult, and often yields little insight. Many details of the hardware architectures and how they affect performance are unknown due to proprietary designs, making the task of separating the effects on performance of the actual code from the effects of the hardware daunting or even impossible. For example, understanding how the network topology and hardware characteristics interact with the MPI implementation and the actual code and its communication patterns is a very difficult task [10].

Tools such as HPCToolkit [11],TAU [12], and Score-P [13] gather different metrics such as execution time, number of floating point operations performed and many more while the application is running. These measurements can be either analyzed in real time with tools such as Periscope [14] or post-mortem, after the application has finished its run. The measurements gathered at runtime can be either summarized in performance profiles, or each separate measurement can be stored individually to create a trace of all events that occurred during a run and their corresponding performance measurements along with their respective timestamps. Traces offer much more information, but require more storage space and incur a much larger overhead (if the gathered data must be written to disk) than creating performance profiles.

1.3.2 Simulations

A more abstract approach simulates program execution with different degrees of accuracy. This allows the developer to understand and quantify the effect of the simulated architecture on performance. The issue with this approach is that accurate simulations, such as cycle-accurate CPU simulations or network simulations are multiple orders of magnitude slower than actually executing the code, while fast simulations run the risk of not representing the real scenario accurately enough to allow informed optimization decisions. Simulations provide vast amounts of data that must be organized and interpreted correctly to yield the desired insights, further complicating the task of obtaining useful results. Simulations can be used for example to perform from cycle-accurate analysis of codes [15, 16] or analyze network behavior with tools like SimGrid [17], DIMEMAS [18], or PSINS [19].

1.3.3 Analytical modeling

Another way of identifying performance bottlenecks such as scalability issues is through analytical performance modeling. In this approach, performance is defined through purely analytical expressions. For example, an analytical scalability model expresses the execution time or other resources needed to complete the program as a function of the number of processors. Unfortunately, the laws according to which the resources needed by the code change as the number of processors increases are often laborious to infer and may also vary significantly across individual parts of complex modular programs. This is why analytical performance modeling — in spite of its potential — is rarely used to predict the scaling behavior before problems manifest

themselves. As a consequence, this technique is still confined to a small community of experts. The results of this type of analysis can be useful in understanding and improving the performance of applications and systems however, as shown by the studies of Kerbyson et al. [20], Mathis et al. [21], Hoefler et al.[22].

If today developers decide to model the scalability of their code, and many shy away from the effort, they first apply both intuition and tests at smaller scales to identify so-called *kernels*, which are those parts of the program that are expected to dominate its performance at larger scales. This step is essential because modeling a full application with hundreds of modules manually is not feasible. Then they apply reasoning in a time-consuming process to create analytical models that describe the scaling behavior of their kernels more precisely. In a way, they have to solve a chicken-and-egg problem: to find the right kernels, they require a pre-existing notion of which parts of the program will dominate its behavior at scale — basically a model of their performance. However, they do not have enough time to develop models for more than a few pre-selected candidate kernels, inevitably exposing themselves to the danger of overlooking unscalable code.

Not only which parts of an application are modeled, but also when in the development cycle performance models are created affects the impact these models have. Access to performance models in early stages of the development process is an indispensable prerequisite to ensure early and sustained productivity. This in turn, requires the availability of continuously updated performance models reflecting design updates and supporting the optimization process. Such models allow problems in applications to be detected early and their severity to be determined when the cost of eliminating the problems is still small. This is increasingly important since mitigating such problems can often take several person years. Despite this, though, current practice often looks different: since creating detailed analytical models is too time consuming, such models are often only built on back-of-the-envelope calculations, rough estimates, simple and manual spreadsheet calculations, or even only developer intuition.These approaches can be misleading and their accuracy is hard to quantify, making their usefulness questionable.

Furthermore, to ensure that applications are performance-bug free, it is often not enough to analyze any one aspect such as processor count or problem size. The effect that the one varying parameter has on performance must be understood not only in a vacuum, but also in the context of the variation of other relevant parameters, including algorithm variations, tuning parameters such as tiling, or input characteristics. This means that any analysis must either make simplifying assumptions potentially introducing inaccuracies in the results, or overcome the challenge of modeling many parameters at once. Furthermore, considering too many parameters makes resulting models hard to understand and limits their practical use.

An application of performance modeling is co-design. Given the tremendous cost of HPC systems, their architectures must match the requirements of the applications they are supposed to run on as precisely as possible [23]. Conversely, applications must be designed such that building an appropriate system becomes feasible, motivating the idea of co-design [1]. In this process, a fundamental aspect of the application requirements are the rates at which the demand for different resources grows as a code is scaled to a larger machine. However, if the anticipated

[1] https://www.anl.gov/articles/co-design-centers-help-make-next-generation-exascale-computing-reality

scale exceeds the size of available platforms this demand can no longer be measured. However, creating analytical models to predict these requirements is often too laborious — especially when the number and complexity of target applications is high.

1.4 Empirical performance modeling

Combining ideas from analytical modeling and experimentation, empirical performance modeling attempts to derive analytical expressions that describe execution time or resource usage from the analysis of experimental measurements. Instead of trying to mathematically model the effect a parameter such as the number of processes has on the execution time of an application, a set of measurements is gathered where the number of processes is varied from run to run. These measurements would then be used to create the desired analytical performance models.

There are multiple challenges when attempting to create empirical performance models. Empirical performance models must be able to represent large and complex parameter spaces, a challenge analytical models also face. The number of measurements that can be performed is often limited by the computational budget of the developers. The number of measurements available is a bigger constraint the more parameters are considered, as the effect each has on performance must be detected. The representation of performance models determines how many behaviors they can express and therefore how useful and accurate they can be, but also how the space of all possible performance models can be searched and how expensive the search is. For example, any set of n measurement points can be trivially fitted by a polynomial of the n-th. order, but such a representation is unlikely to accurately model the behavior observed. Finding models which accurately explain measurements without including noise in the results is difficult, and in some cases even quantifying the noise is not a trivial task. Finally, gathering performance measurements and manually creating performance models from them is possible, if time-consuming, when considering whole applications. However, if individual kernels are considered the effort would certainly be untenable.

1.5 Contributions

We believe it is important that instead of modeling only a small subset of the program manually, performance models are made available for each part of a program automatically, significantly increasing not only the coverage of the performance check but also its speed. Using detailed empirical measurements as a starting point, this approach would permit performance engineers to focus on problematic kernels without sacrificing coverage.

The goal of this work has been to find an automated method capable of representing the compounded effects of a set of one or more parameters on performance simultaneously and overcome the challenges of empirical performance modeling. Furthermore, we introduce a method of applying the modeling approach to support requirements engineering and co-design processes. The performance modeling approach is implemented in the Extra-P [2] analysis framework, providing a library for easy integration with other tools and applications, as well as a

[2] http://www.scalasca.org/software/extra-p/download.html

graphical user interface to visualize and explore resulting performance models. The specific contributions are briefly described below:

Automatic performance modeling. The first step towards our goal is to generate performance models for one parameter, such as process count or problem size using as few as five measurements. To represent performance models in a flexible, but simple and intuitive way, we introduce the Performance Model Normal Form (PMNF) [24]: We take advantage of the observation that the space of the function classes underlying performance models is usually small enough to be searched by a computer program. We generate a search space out of possible complexity classes represented by combinations of polynomial and logarithmic terms. For example, the performance model describing execution time for sorting a set of n items as a function of n could be represented as $t(n) = 0.1 \cdot n^2$ for an unoptimized implementation. We propose an iterative refinement process that maximizes both the efficiency of the search and the accuracy of generated models. We leverage the Score-P measurement infrastructure to create performance models at the granularity of function calls. On one hand, this provides detailed performance information and identifies critical code regions. On the other hand, the fine granularity translates into mostly simple models, which makes both the generation faster and their subsequent evaluation simpler. We create requirement models alongside execution-time models. A comparison between the two can illuminate the nature of a scalability problem.

Multi-parameter performance modeling. As the compounded effect of multiple configuration parameters is very important for many applications, in the next step we extend the automatic performance modeling method to quickly create empirical performance models of parallel applications as a function of a small arbitrary set of input parameters. The main challenge when considering multiple parameters consists of the search space explosion as all possible combinations of polynomial and logarithmic terms for each parameter must be analyzed. We propose two heuristics to accelerate the search for suitable performance models employed by our generation method, making the approach practically feasible [25]. The first heuristic speeds up multi-parameter modeling, as it reduces the search space to only combinations of the best single-parameter models. The second heuristic speeds up model selection for single parameter models. Combined, the heuristics allow a search space of hundreds of billions of models to be reduced to under a thousand and to reduce the time required to obtain a multi-parameter model from over six years to under six milliseconds.

Requirements engineering using performance modeling. We propose a method to study the design of parallel systems based on the established requirement models and leveraging the capability to model the compounded effects of problem size per process and number of processes on the different requirements. The generation of requirement models is less affected by performance variations, and focuses on what the application required from the hardware, rather than specific features or limitations of the hardware itself. For example, we consider the number of floating point operations an application requires to solve a particular problem, rather than the time it for those operations to be performed on a particular system. We focus on comparing different design decisions such as the trade-off between the number of processors

on a system versus the floating point operations per second rates each processor provides. We therefore offer a technique for practical co-design: the requirement models and system study can point out potential bottlenecks very early in applications or systems that either need to be mitigated or could result in substantial cost savings if they are mitigated.

Beyond the contributions mentioned above, the approaches developed in this thesis had an impact on the work of other researchers. Collaborations with other authors have improved and expanded the capabilities of Extra-P [4, 6]. Extra-P is used by application developers and performance engineers to understand the performance of scientific codes and libraries, such as UG4 [7], FASTEST-3D [26], and OpenMP [5]. New approaches in analyzing the performance of HPC applications and systems have been developed in collaboration with other researchers using Extra-P as a starting point [3, 9].

1.6 Structure of this document

The rest of this document is structured as follows: First,we instroduce the automatic empirical performance modeling method in Chapter 2. Then, we expand the approach to cover multi-parameter models in Chapter 3. In Chapter 4 we apply our method to perform requirements engineering. Research avenues mode possible by this work are presented in Chapter 5. The open-source tool Extra-P which implements the methods and approaches introduced in this work is shortly described in Chapter 6. We compare our contributions to other approaches in the field in Chapter 7. Finally, we present the conclusion and outlook in Chapter 8.

2 Automatic Performance Modeling

The first contribution of this work is a method to identify *performance bugs* with a particular emphasis on *scalability bugs*. A scalability bug is a part of the program whose scaling behavior is unintentionally poor, that is, much worse than expected. As computing hardware moves towards exascale, developers need early feedback on the scalability of their software design so that they can adapt it to the requirements of larger problem sizes on bigger machines. This method can be applied to both strong- and weak-scaling codes. Extra-P is the open source performance modeling tool implementing the methods introduced in this thesis. In addition to searching for performance bugs, the models Extra-P produces also support projections that can be helpful when applying for the compute time needed to solve the next larger class of problems. Finally, because Extra-P models both execution time and requirements alongside each other, the results can also assist in software-hardware co-design or help uncover growing wait states.

The input of Extra-P is a set of performance measurements where a given parameter is varied. The purpose of the analysis is to model and quantify the effect that this variation has on application performance by analyzing relevant measurable metrics such as execution time, number of floating-point operations, or number of bytes sent over the network. To obtain the scaling models which are a often a focus of this type of analysis, the parameter varied is the number of processes or threads used by a given application, $\{p_1, \ldots, p_{max}\}$, and the performance measurements take the form of parallel profiles. In this case, the output of Extra-P can be summarized as a list of program regions, each described by a function describing measured performance metrics as a function of the number of processes p. This list can then for example be ranked by the predicted execution time of each region at a target scale of $p_t > p_{max}$ processors. More complex ways of sorting and analyzing the results are available, but the approach described above is the most common one requested by developers, as it allows future performance bottlenecks to be easily identified. These regions will be referred to as *kernels* and they define the code granularity at which performance models are generated.

Users who want to know the scalability of an application at much larger scales, such as when preparing applications for the next generation of supercomputers, will likely choose $p_t \gg p_{max}$. In the evaluation in Section 2.7, reasonably accurate projections for $p_t = 128 \cdot p_{max}$ are demonstrated. If only the asymptotic behavior is of interest (i.e., $p_t \to \infty$), the ranking can be based exclusively on the growth function class itself. The goal is not 100% accuracy. Such a claim would be impossible due to the results being based on empirical measurements —especially when the ranking is based on the times predicted for a specific scale $p_t \gg p_{max}$. However, it will usually be good enough to draw attention to the right kernels. Of course, false negatives, which are program regions that are not identified because they wrongly appear too far at the bottom, may occur if a phenomenon relevant at scale is not captured in the input data. Nevertheless, given that confidence information is provided along with the models, false positives should be extremely unlikely. In a final step, the user needs to compare the projected with the expected

behavior for each kernel. This has to be done manually because user expectations cannot be predicted nor can it be assumed that the user has precise expectations for every kernel identified. To formulate expectations users may, for example, rely on the isoefficiency metric [27].

In general, the underlying mathematical framework can accommodate more or simply different independent parameters than just the number of processors p. For example, Section 2.7.2 shows how to obtain highly accurate performance predictions when varying the problem size per process while keeping p constant. Nevertheless, this part of the work puts the emphasis on varying p only, while assuming that all other input parameters either depend on p or remain stable. Note that violations of this assumption do not preclude the application of the method, they simply lower the accuracy with which performance bugs can be identified.

2.1 Performance model normal form

Creating the model describing the effect of varying a parameter on a chosen metric as a function forms the core of our method. Therefore, we start by explaining the different aspects of this approach and their underlying ideas in detail.

When generating performance models, we exploit the observation that algorithms have certain commonalities when considering their behavior as a function of one or more variable performance-relevant parameters. Most behaviors can be defined as a finite number n of predefined terms, involving powers and logarithms of the chosen parameters. In the case of scaling behavior, the parameter would be p, leading to the following expression:

$$f(p) = \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot log_2^{j_k}(p) \tag{2.1}$$

This representation is, of course, not exhaustive, but works in most practical scenarios since it is a consequence of how most computer algorithms are designed. We call it the *performance model normal form* (PMNF). While algorithms can be constructed with more exotic functions, such as $sine$ or factorial progressions, they are far from common. Or approach can be easily adapted to include other types of function classes such as logarithms of logarithms or exponential functions, but in our experience, they are too seldom encountered to warrant a significant increase in the time needed to model the common case. We base not just on own experience with HPC applications but also on the complexity classes of the 13 dwarfs of parallel computing[28].

Moreover, our experience suggests that neither the sets $I, J \subset \mathbb{Q}$ from which the exponents i_k and j_k are chosen nor the number of terms n have to be arbitrarily large or random to achieve a good fit. Thus, instead of deriving the models through reasoning, we only need to make reasonable choices for n, I, and J and then simply try all assignment options one by one. A possible assignment of all i_k and j_k in a PMNF expression is called a *model hypothesis*. Trying all hypotheses one by one means that for each of them we find coefficients c_k with optimal fit. Then we apply cross-validation [29] to select the hypothesis with the best fit across all candidates. Of course, the computational effort required to calculate our model depends on n, $|I|$, and $|J|$. On the other hand, a larger number n of constituent terms does not necessarily imply a better model.

To strike a good balance, our models are generated in an iterative refinement process, which we outline in Section 2.3.4. To cover most common complexity classes encountered, we select $n = 2$, $I = \left\{\frac{0}{4}, \frac{1}{4}, \ldots, \frac{12}{4}\right\}$, and $J = \{0, 1, 2\}$ as a default. Given that the tuples $(i, j) \in I \times J$ can be ordered by their corresponding asymptotic behavior, our choices for I and J reflect a range of behaviors from perfect to poor scalability in 39 steps. Scalability worse than $p^3 \cdot log^2(p)$ is not distinguished. If the behavior of the application is already known to some degree, the sets I and J can be extended to provide more detail in a given range. For example, adding more fractional exponents in the (0,1) interval such as $\left\{\frac{1}{3}, \frac{2}{3}, ...\right\}$ for applications where the goal is not to find out whether they scale at all but rather how well they scale can provide additional insight.

2.2 Parameter space exploration

Two aspects are critical to obtaining meaningful performance experiments, namely which parameters are analyzed, and for each parameter, which range of values is investigated. The costs of running performance experiments often makes an exhaustive search of all parameters and value ranges impossible and therefore a careful selection must be made.

2.2.1 Parameter selection

The parameter most commonly varied in our studies is the number of processors used. Both strong scaling, where the overall problem size is kept constant, and weak scaling, where the problem size per process is kept constant are often encountered and our approach can handle both with ease.

We have discovered that if we can define an expected ideal invariant for application behavior with respect to a given parameter, it becomes much easier for users to understand and interpret the resulting models. Using the previous example, the invariant for strong scaling is the total work which should ideally stay constant. By gearing the metrics we use as an input towards this invariant, most resulting models are constant as they incur no communication or synchronization overhead. As an example, for strong scaling studies we model the total runtime summed up across all processes. Similarly, for weak scaling studies we model the runtime per process, averaged across all processes.

For other parameters, such invariants could be helpful if they are determined. However, their effects are often contained to only a small number of kernels, therefore making the interpretation of results less time consuming. For example, when considering problem size determining the ideal application behavior is clearly not possible in a general case.

Another type of parameter which has a significant effect on performance is the problem size. It can take different forms for different applications, such as the size of a mesh or number of particles. The problem size can sometimes be composed of different independent parameters, such as the length, width, and depth dimensions of a three-dimensional volume. The effect of each individual parameter can be analyzed and often parameters can have varied effects on application performance. Due to how the code is implemented, this can happen even if the

parameters should intuitively have the identical effects. For example, the ordering of nested loops over different dimensions and its effect on cache behavior can lead to these parameters having different effects performance.

Other parameters can be encountered but are often applications specific. However, as long as their effect on performance can be represented by the PMNF, the resulting models will yield accurate predictions and offer insights to the developer.

2.2.2 Parameter value selection

Extra-P uses an empirical approach to performance modeling and attempts to create insightful performance models with as few as five different performance experiments. Therefore, how the performance experiments are selected will have a strong impact on the accuracy of the resulting models. Some applications impose restrictions on what values parameters can take, such as requiring the processor count to always be a square or a cube number.

The most important lesson is that as an empirical method our approach can only model performance effects present in the data. For example, if all scaling experiments are performed within a node on a supercomputer, the effects that the inter-node network has on performance will not be captured. This can potentially lead to inaccurate models when attempting to predict and understand bottlenecks on the scale of the full supercomputer. We recommend the developers to consider the architecture they are performing their experiments on, and selecting parameter values such that the behavior they wish to model is represented within the value range chosen. As a rule of thumb, scaling experiments should be performed on the smallest range of processor counts that encompasses all behaviors that must be understood to define application performance. In the scaling example before, the experiments should include runs using multiple racks, therefore encompassing both inter- and intra-node communication.

Another consideration for value selection is that all values should represent the same type of behavior. For example, many supercomputers and MPI libraries function qualitatively different if the processor count is a power of 2. Therefore, in such cases processor counts should either be chosen to all be powers of 2 or none of them.

2.3 Workflow

Figure 2.1 gives an overview of the different steps necessary to find scalability bugs, whose details are explained further below. To ensure a statistically relevant set of performance data, profile measurements may have to be repeated several times—at least on systems subject to jitter. This is done in the optional statistical quality control step. Once this is accomplished, regression is applied to obtain a coarse performance model for every possible program region. These models then undergo an iterative refinement process until the model quality has reached a saturation point. To arrange the program regions in a ranked list, the performance is extrapolated either to a specific target scale p_t or to infinity, which means the asymptotic behavior is used as the basis of the comparison. This is achieved by comparing the exponents of the resulting models. As we only produce terms containing only polynomial and logarithmic terms,

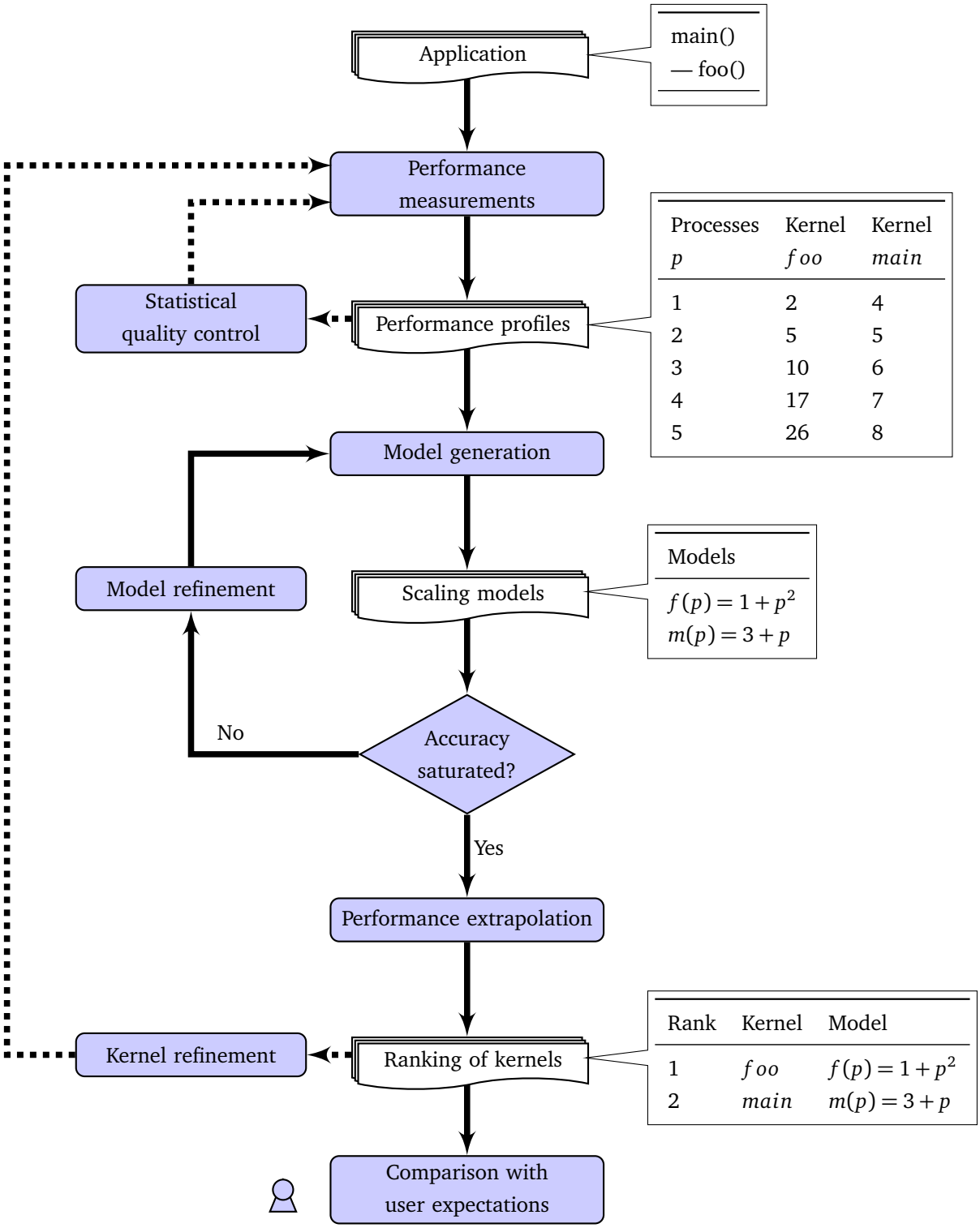


Figure 2.1: Workflow of scalability-bug detection. Solid boxes represent actions or transformations, and banners their inputs and outputs. Dashed arrows indicate optional paths. A representation of the different results is provided for a minimal example in callouts on the right side of the figure.

creating a total asymptotic ordering is trivial. Finally, if the granularity of our program regions is not sufficient to arrive at an actionable recommendation, performance measurements, and thus the kernels under investigation, can be further refined via more detailed instrumentation.

2.3.1 Performance measurements

We generate the parallel profiles needed as input to our tool using Score-P [13], a measurement infrastructure that is highly scalable and can be used for profiling, event tracing, and online analysis of HPC applications. Score-P records the execution time plus various performance counters, including hardware counters such as the number of floating-point instructions and software counters such as the number of bytes an MPI function sends or receives. All metrics are broken down by call path and process. We define a call path as a program region plus its calling context such as $main \rightarrow foo \rightarrow MPI\_Send$. Going beyond purely static program regions will allow scalability problems to be pinpointed more precisely. To keep the code covered by a call path small, performance metrics are collected with exclusive semantics, that is, for each call path without including its children. Score-P has a default instrumentation scheme that delivers performance data at the granularity of functions as call-chain elements. However, manual instrumentation can be added to distinguish lower-level constructs such as loops, which may be needed during kernel refinement. In any case, at the default granularity, computational call paths are already clearly distinguished from communication, ensuring that communication is modeled separately from computation. In the next step, we collapse the process dimension via median reduction, keeping one value per call path and metric. The target function we want to model is thus the median wall-clock time per call path. We could also choose the maximum to enables us to capture bottlenecks even if they are confined to a small subset of the processes. However, the the maximum is more sensitive to statistical outliers and may pose the risk of reducing the model quality. Of course, aggregation across all processes assumes bulk synchronous parallel (BSP) programs. For irregular applications, such as some graph computations or task-based execution programs, we would have to resort to a hierarchical scheme that only summarizes subsets of processes with similar behavior. Of course, this would increase the number of performance models. An approach towards the performance modeling of task-based applications has been developed and is described in more detailed in [9].

The metrics we collect include both requirements-based metrics and time-based metrics. Requirements-based metrics such as the number of arithmetic operations or the number of messages sent or received are usually a function of the execution configuration and therefore more or less deterministic. As a welcome side effect, they are largely immune to system noise [30]. We call them *requirements based* because they reflect the requirements of the program rather than the resources mustered to satisfy them. Requirements-based metrics play an important role in our approach because—supported by their deterministic nature—they can often be used to determine the function class underlying our performance models more easily. Frequently, this function class is known a priori. Time-based metrics, such as the wall-clock time spent in communication, in contrast, are needed to determine the coefficients of our model functions or the model functions themselves when they cannot be expressed as a function of requirements-based

metrics. We discuss discrepancies between time-based and requirements-based models later in Section 2.3.3. In any case, time-based metrics are indispensable when we want to extrapolate execution times to a specific p_t.

2.3.2 Statistical quality control

On many systems, performance measurements are subject to serious run-to-run variation as a consequence of OS jitter, network contention, and other nondeterministic factors. Although not an intrinsic element of our approach, we anticipate such noise and account for it by calculating confidence intervals. For this purpose, the user can repeat measurements until the variance stabilizes. Then, Extra-P performs a check to ensure that the deviation is not prohibitive. An extreme example of such a case would be a system where the deviation across repeated measurements with the same input configuration is greater than the difference across different configurations, rendering any subsequent modeling meaningless.

2.3.3 Model generation

Model generation forms the core of our method. The Performance Model Normal Form, introduced in Section 2.1 is used to create a search space from which models are selected. By trying out all model hypotheses generated by given subsets $I, J \subset \mathbb{Q}$ from which the exponents i_k and j_k are chosen and a number or terms n we can select the hypothesis which best fits the measured data.

To measure the fit we use cross-validation. This involves dividing the performance data into training and evaluation sets (i.e., sets of profiles). We use the training sets to create the model and the evaluation sets to calculate the fit. This has the advantage of protecting against overfitting, which may result in a model that tightly fits the input data points but does not accurately represent the asymptotic behavior. This problem is often encountered when fitting a polynomial of an order higher than or equal to that of the number of available data points. Specifically, we apply k-fold cross-validation, including the variants of holdout and leave-one-out [29, 31, 32, 33]. k-fold cross-validation divides the input into k sets of equal size. One of the sets is used for validation and all others for training. The holdout method divides the data into two sets of equal size ($k = 2$), using one as the training and the other as the evaluation set. The leave-one-out-cross-validation (LOOCV) method uses, as its name suggests, a single data point (i.e., profile) for validation and all others for training. LOOCV is the slowest because it requires as many cross-validation passes as there are data points. On the other hand, it provides good results for very small numbers of data points (< 10). K-fold is faster but requires more data points. We implemented the general algorithm allowing k to have arbitrary values. Our experiments suggest that the holdout method delivers the best time-accuracy tradeoff and as such we propose $k = 2$ as a default, but we provide the option of changing it to suit the user's needs. When creating the sets we assign adjacent data points to different sets.

We use the residual sum of squares, a quality-of-fit metric, as an indicator for the quality (and thus confidence) of our model. For a fit of n variables with measurement values y_i and fitted

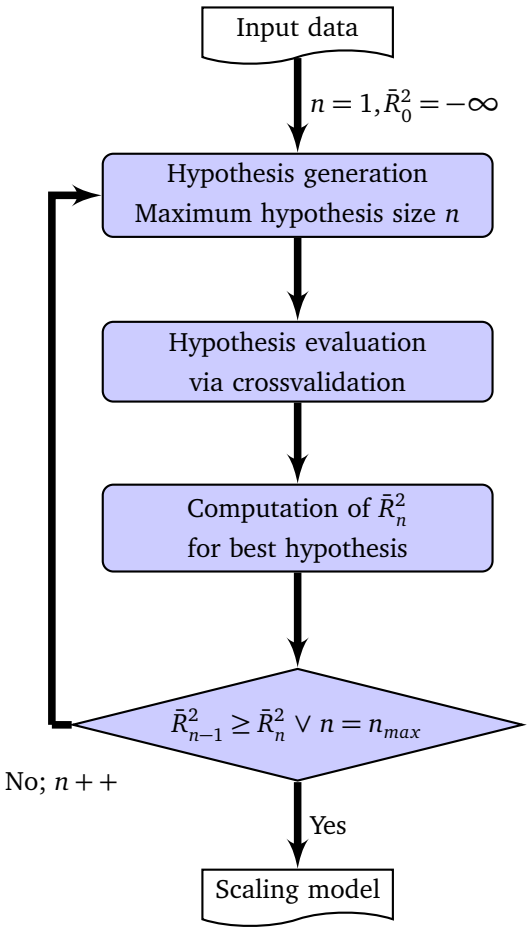


Figure 2.2: Iterative model refinement process. Solid boxes represent actions or transformations, and banners their inputs and outputs.

hypothesis model $f(x_i)$ $(1 \leq i < n)$, $RSS = \sum_{i=1}^{n}(y_i - f(x_i))^2$. We calculate the coefficient of determination $R^2 = 1 - \frac{RSS}{TSS}$ as a measure of fit, where $TSS = \sum_{i=1}^{n}(y_i - \bar{y})^2$. If $R^2 = 1$, the model fits the data exactly.

2.3.4 Model refinement

To arrive quickly at a suitable model hypothesis and to protect against overfitting, we start with a coarse approximation that we successively refine until we reach the point of *statistical shrinkage* [34]. This is the point after which the model starts to lose predictive power outside the range of samples. The whole refinement process is summarized in Figure 2.2. At the beginning, we allow a maximum of one term in our hypothesis chosen via cross-validation, as described earlier. We then compute the adjusted coefficient of determination $\bar{R}^2$ [34] of the best model we can find. In the next iteration, we allow a maximum of two terms in the hypothesis. We repeat the cross-validation to find the best model and compute the new adjusted $\bar{R}^2$. If the new value computed for the adjusted $\bar{R}^2$ is smaller than the previous one, indicating that adding more terms to the hypothesis would lead to statistical shrinkage, then the iterative process stops. Otherwise, we continue adding terms to the model until we reach either the shrinkage

point or a configurable limit, which may be the maximum time we are prepared to invest or the maximum number of terms we want to consider.

We have discovered however that the benefits of models refinement are mostly usable on counted metrics where the noise is very small or in cases where a lot of measurement points are available. If less than 10 distinct data points are available, allowing more than two non-constant terms for performance models almost always leads to poor models that model noise. Furthermore, this approach cannot adequately compare models with degrees of freedom to the constant model, which is a case that repeatedly comes up in practice. Another approach will be described in Section 5.2 that was developed to combat this issue.

2.3.5 Performance extrapolation

Once we have reached this point in our workflow, we have a model describing the scaling behavior of each call path in our application. Now, we can evaluate the scaling function for a target scale p_t or just look at the asymptotic behavior. We can either extrapolate execution time or requirements (e.g., bytes sent/received or floating-point operations). The latter can also be helpful in finding roofline models [35], which take resource limitations into account that become effective only at larger scales, an extension of our method discussed in more detail in Chapter 4. Extrapolating requirements is also relevant to system design because it allows the hardware resources to be optimally balanced according to an application's future needs. Of course, the model we create can only reflect information and phenomena present in the data. As such, any projections will not account for effects that only come into play outside the scope of the experiments. A simple example is the change in algorithm some MPI collective operations perform when certain process numbers are exceeded. Unless this occurs within the experimental data the method will not predict its effect on performance. Regardless of whether the user chooses a specific target scale p_t or is just interested in the asymptotic behavior, we are now in a position to rank all call paths by their expected performance impact. Those at the top of the list are the kernels whose models the user should compare to his or her expectations and analyze further if serious discrepancies arise.

2.3.6 Kernel refinement

Once the kernels relevant at the target scale have been determined, the user may find the granularity of these kernels too coarse and, as a consequence, the resulting performance model too complex to draw meaningful conclusions. This can happen if the default instrumentation of Score-P, which is typically applied at the level of functions, is not fine-grained enough to pinpoint pieces of code that are small enough for inspection by a human user. In this case, the instrumentation around the kernels of interest can be narrowed or the kernels split into multiple pieces to be modeled separately. At the same time, the instrumentation around those parts of the program that our analysis classifies as irrelevant can be lifted to reduce instrumentation overhead. Then the whole process starts over again: this time with more targeted measurements that exploit the knowledge gained in the previous iteration.

2.3.7 Example

To summarize the workflow we offer a very simple example of an application with just two kernels, $main$ and foo, shown in Figure 2.1. The user first runs five performance experiments, wherein the number of processes is varied from one to five. The tool models each kernel separately and determines the models $f(p) = 1 + p^2$ for foo and $m(p) = 3 + p$ for $main$. The kernels, ranked by their asymptotic growth, are then presented to the user.

2.4 Modeling requirements alongside time

Another key aspect of our approach is that we build requirements models alongside execution-time models and compare them to each other. In essence, we build empirical requirements models, which we subsequently try to match with the measured execution time. As we will explain further below, the quality of this match can reveal important facts about the application—regardless of whether the models are in agreement or show discrepancies. Since requirements-based metrics are much less prone to jitter than time-based metrics, they are much more likely to capture the asymptotic behavior correctly. This is because requirements models are closer to algorithmic complexities than empirical models derived exclusively from time-based metrics. In fact, an empirical requirements model alone can provide valuable insights when compared to developer expectations. The general idea of a requirements model is to account for all major cost factors. For computational call paths, these are the different types of operations such as floating-point operations, loads, stores, etc. Of course, in empirical models these operations have to be mapped onto the instruction set and the hardware counters available on the target system. For communication call paths, the cost factors are the number and the size of messages. We measure them using the standardized PMPI interface [36], which is portable across all MPI implementations.

We try to choose our metrics such that each cost factor is counted only once, although a certain degree of overlap can be tolerated. For the sake of simplicity, we assume that the total costs within a certain cost category (e.g., floating-point operations) rise linearly with the corresponding metric. That is, twice as many operations will take twice as long. Of course, some costs may disappear through latency hiding, prefetching etc. Nevertheless, we believe that this inaccuracy matters much less when our primary question only refers to the behavior at scale. Taken to an extreme, the asymptotic complexity of the scaling function does not improve simply because we can execute four floating-point operations and two loads or stores at once.

To express the execution time of a call path as a function of its requirements, we distinguish between local and global operations. Recall that the maximum aggregation across all processes we perform essentially results in a process-local metric. For the execution time of local operations, which cover computation and point-to-point communication, we therefore assume a linear relationship. This is because all processes can carry out their local operations in parallel. For example, the time it takes to send a certain number of messages m of size s is $t = m(c_1 + c_2 \cdot s)$. Of course, the number of messages a process sends may depend on the input configuration.

Thus, for a given call path, we model each of the requirements-based metrics separately, generating a full regression model for each metric. Now, we compare each requirement model with the model for execution time. If they are in good agreement, the user can draw conclusions about the primary factor contributing to the time (e.g., number of messages or floating-point operations). If not, the user can regard this as a sign that the execution time does not exclusively depend on the requirements and may be prolonged by wait states. An extreme example is serialized code. There, the waiting time dilates both mean and overall execution times linearly with p. Another example is collectives whose execution times are extended by jitter [30]. A very common source of wait states is load imbalance, a problem that particularly affects irregular codes such as implementations of climate simulations [37].

2.5 Effort

The required computational effort consists of two components—running the input experiments and running the model generator. As long as only one model parameter is used, the latter takes less than a minute on a single processor and is therefore negligible. The cost of the input experiments can be quantified in relation to experiments at the target scale, which our method helps to avoid.

In weak scaling mode, the compute time of a perfectly scaling code in node hours is proportional to the number of processors. Assuming that the number of processors is always a power of two, running experiments at input scales $\{2^0, \ldots, 2^m\}$ together is thus less expensive than a single run at $p_t = 2^{m+1}$. If the code scales poorly or the target scale grows beyond 2^{m+1}, the amortization factor can increase substantially. Jitter may require more experiments per input scale, but to be conclusive experiments at the target scale would have to be repeated as well.

Strong scaling experiments should have, barring overheads, the same computation costs. Here, avoiding experiments at larger scales does not reduce the effort, but provides the opportunity to gain insights into how an application would perform at a larger scale, even if a larger system is not available.

2.6 Evaluation with synthetic data

To evaluate our approach we will first test the performance model generator on input sets originating from known underlying functions, using a methodology proposed by Reisert et al. [4]. To achieve this, we generate functions using the following classes of terms as building blocks:

- Constant
- Common: x, x^2, x^3, $\log_2(x)$
- Rare: $x^{\frac{i}{2}}$ for $i \in \{1,3,5\}$, $x^{\frac{i}{3}}$ for $i \in \{1,2,4,5,7,8\}$, $\log_2^2(x)$
- Exotic: $x^{\frac{i}{4}}$ for $i \in \{1,3,\ldots,11\}$, $x^{\frac{i}{5}}$ for $i \in \{1,\ldots,14\} \setminus \{5,10\}$, $\log_2^{\frac{1}{2}}(x)$, $\log_2^{\frac{3}{2}}(x)$

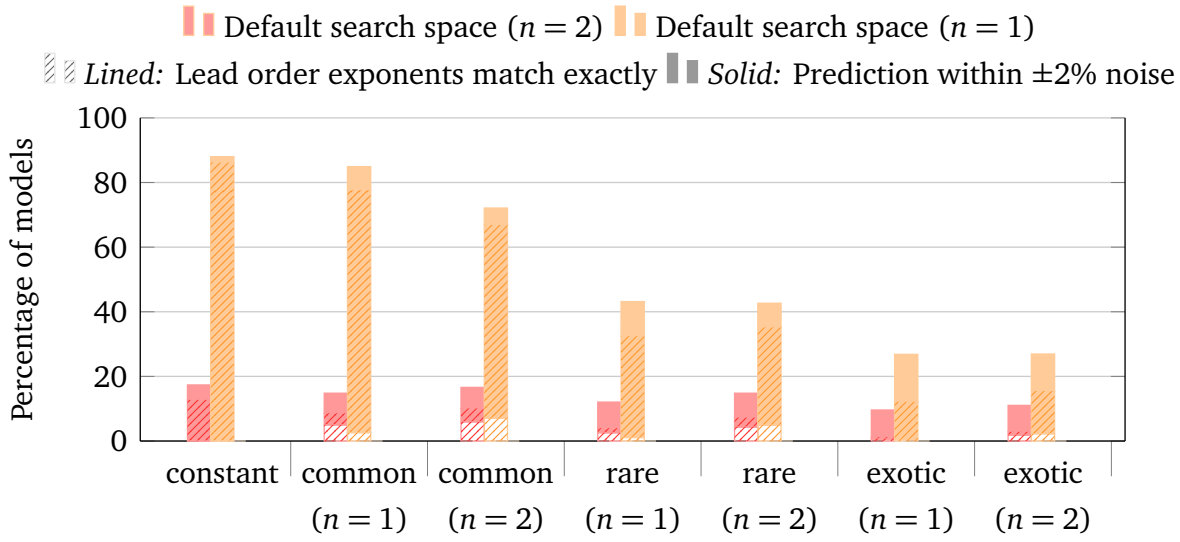


Figure 2.3: Comparison of evaluating the approach with a maximum of either a single term or two terms per model. We are using values of randomly generated functions with ±2% of noise as input. The functions are built according to the PMNF with $n = 1$ or $n = 2$, and their coefficients c_0, c_1 and c_2 are calculated by sampling $a \in [-2, 3]$ uniformly and then computing 10^a.

Our experience has shown that most functions encountered are constant with respect to the analyzed parameters. Most functions that show a non-constant behavior can be described only using common terms or combinations thereof, and correspond the the asymptotical complexity of most classical algorithms. Functions containing terms from the rare class have been encountered by us in practice but only in a handful of situations such as geometric effects of dividing processors across two or three dimensions in an even manner. In the exotic category, we find terms that while theoretically possible have yet to be encountered by us in a real scenario. For all non-constant cases we generate test functions with one or two terms.

For each class described, we generate 1000 functions. We randomly generate a constant term by sampling $a \in [-2, 3]$ and computing 10^a. The coefficients of other terms are generated in a similar manner as necessary. These functions are then evaluated for each of the following four different sets of x values ({2, 4, 8, 16, 32}, {8, 16, 32, 64, 128}, {32, 64, 128, 256, 512}, and {128, 256, 512, 1024, 2048}) to create inputs for the model generator. Finally, random uniform noise of up to plus or minus 2% of the evaluated value is applied to each function value.

We wish to understand if our method is capable of identifying the asymptotic behavior correctly. Furthermore, we also wish to know if the determined model is capable of predicting the function value for an x four times larger than the larger evaluated input within the same noise tolerance of plus or minus 2%. We have chosen a factor of four to show the predictive capacity of this modeling approach.

Figure 2.3 shows that our approach correctly identifies the asymptotic trend and even correctly predicts behavior outside the measured interval in more than 75% cases for the constant and common cases. This test further proves that our assumption that five data points are sufficient to allow accurate modeling of one term to be correct. However, attempting to model two

terms with only five data points will usually fail, cementing the idea that only the main trend should be identified.

2.7 Case studies

We illustrate the capabilities of our tool using five MPI applications. Specifically, we demonstrate that our tool

- identifies scalability issues in codes known to have them,
- does not identify a scalability issue in codes that are known to have none
- identifies previously unknown scalability issues

We find the models we generate automatically to be in good agreement with manually created models previously reported in the literature. In one of the case studies we further show that we can produce accurate models for model parameters other than the number of processes.

We performed our experiments on the IBM BlueGene/Q system Juqueen and the Sun cluster Juropa at the Jülich Supercomputing Centre. Juqueen is a large leadership supercomputer with almost 500,000 cores. Each node features one PowerPC A2 processor with 16 cores running at 1.6 GHz. Juropa is a compute cluster composed of 2,208 nodes, each equipped with two Intel Xeon X5570 (Nehalem-EP) quad-core processors running at 2.93 GHz. Unless otherwise stated, we always used the default settings for n, I, J specified in Section 2.3.3. We ran the model generator on several desktop systems and front-end nodes, where model generation for a single but full code never exceeded one minute.

2.7.1 SWEEP3D

In this example, we show how our tool helps identify and explain a scalability problem, providing a first impression of the user experience. The Sweep3D benchmark [38] is a compact application that solves a 1-group time-independent discrete ordinates neutron transport problem. It was extracted from a real ASCI code. The program calculates the flux of neutrons through a three-dimensional grid along several angles of travel. To partition the problem, the code maps the three-dimensional domain onto a two-dimensional grid of processes. Parallelism is achieved through a pipelined wavefront process that propagates data along diagonal lines through the grid. The particular angle being processed at a given moment determines the direction of the wavefront, which can originate from any of the four grid corners. The pipeline organization enables multiple wavefronts to follow each other along the same direction, although the inability of a process to satisfy horizontal and vertical neighbors at the same time introduces propagating delays [39]. Parallel efficiency drops further whenever the pipeline has to be refilled after the direction has changed. In both cases, the consequences are wait states that materialize in receive operations.

Table 2.1 lists the five kernels that would consume most of the time at the target scale $p_t = 262,144$ processes, ranked by their predicted execution time. To underline that indeed the right kernels appear at the top, we show their measured execution time in terms of both their relative

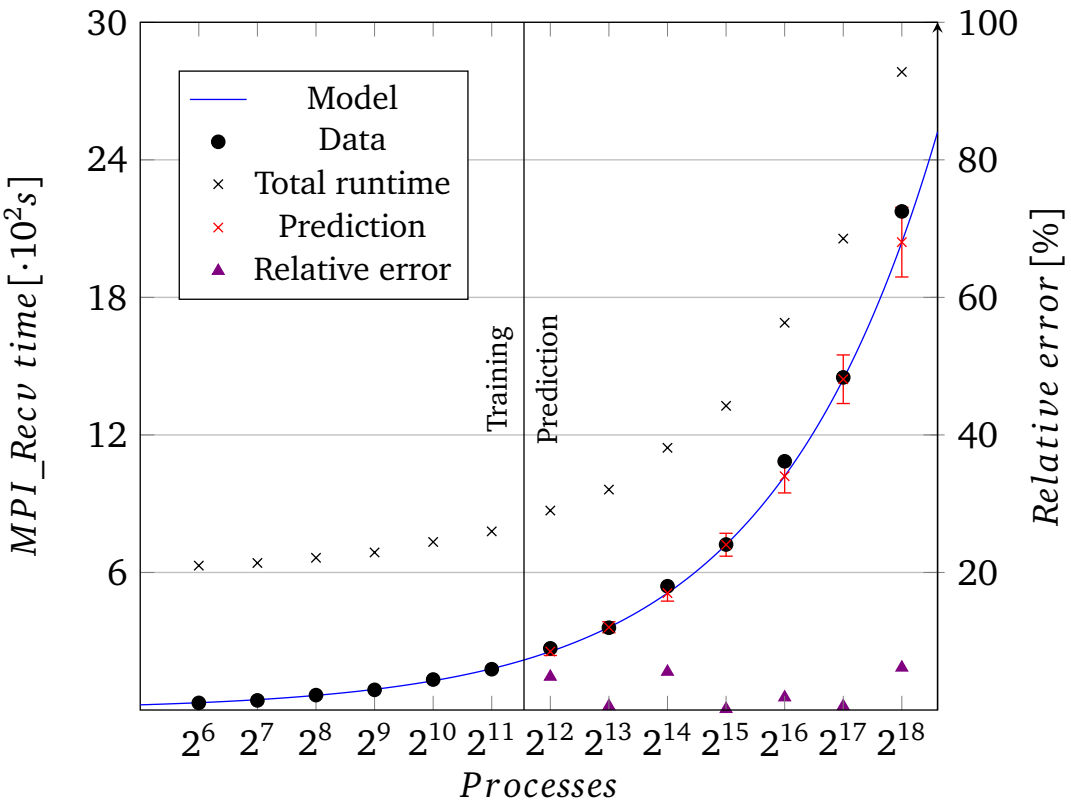


Figure 2.4: Measured vs. predicted execution time of the two receive operations involved in the wavefront process of Sweep3D on Juqueen.

contribution at p_t and the increase factor of their execution time in comparison to $p = 64$. The latter offers some intuition on how seriously the performance is affected. Together, all five kernels account for more than 99% of the overall runtime. Although our predictions based on training data with up to 8k processes are closer to measured values, even predictions based on training data with up to 2k still show the same general trend. In particular, the ranking remains unchanged. Note that adding more data points does not change the model hypothesis for four of the five kernels, only their coefficients vary slightly to reflect the increased precision allowed by the additional training points. The changing model for $sweep \rightarrow MPI\_Send$ reflects that the new training points manifest a new effect which was previously impossible to see at smaller scales. In this specific case, the runs at 2k processes and beyond allow the latency effect of communication leaving the node board to be observed.

The literature mentions accurate models [40, 41] that describe the performance behavior of wavefront processes as they occur in Sweep3D on various architectures. The LogGP model reported in Hoisie et al. [40] characterizes the communication time as follows:

$$t^{comm} = [2(p_x + p_y - 2) + 4(n_{sweep} - 1)] \cdot t_{msg} \tag{2.2}$$

p_x and p_y denote the lengths of the process-grid edges, n_{sweep} the number of wavefronts to be computed, and t_{msg} the time needed for a one-way nearest-neighbor communication. Given

Table 2.1: The most time-consuming Sweep3D kernels (i.e., call paths) ranked by their predicted execution time at the target scale $p_t = 262,144$ processes. The values and models reflect exclusive execution times without callees. The predictive error applies only to p_t. On the left we used training data with up to 2,048 processes ($p_t = 128 \cdot p_{max}$), on the right with up to 8,192 processes ($p_t = 32 \cdot p_{max}$).

Runtime	P_1 ($p_i \leq 2,048$)		P_2 ($p_i \leq 8,192$)	
[%]	Model [s]	Predictive	Model [s]	Predictive
$p_t = 262k$	$t = f(p)$	error [%]	$t = f(p)$	error [%]
sweep → MPI_Recv				
65.35	$3.99 \cdot \sqrt{p}$	6.16	$4.03 \cdot \sqrt{p}$	5.10
sweep				
20.87	582.19	0.01	582.19	0.01
global_int_sum → MPI_Allreduce				
12.89	$0.94\sqrt{p} + 0.04\sqrt{p}\log p$	23.00	$1.06\sqrt{p} + 0.03\sqrt{p}\log p$	13.60
sweep → MPI_Send				
0.40	11.66	29.00	$11.49 + 0.09\sqrt{p}\log p$	15.40
source				
0.25	$6.86 + 9.68 \cdot 10^{-5}\log p$	0.01	$6.86 + 9.13 \cdot 10^{-5}\log p$	0.01

that both n_{sweep} and t_{msg} are largely independent of the number of processes p and that in our experiments $p_x = p_y$ and $p = p_x \cdot p_y$, we can rewrite Equation (2.2) as:

$$t^{comm} = c \cdot \sqrt{p} \tag{2.3}$$

The (combined) model generated by our tool for the two receive operations involved in the wavefront process (sweep → MPI_Recv) is $3.99 \cdot \sqrt{p}$ and, thus, consistent with Equation (2.3). As Figure 2.4 illustrates, it also matches our measurements on Juqueen quite accurately. The two receive operations are modeled together because Scalasca's default instrumentation merges them into one call path. Note that we do not need large application runs to accurately determine the model. The figure presents results based on only six training and evaluation data points with the process counts $P_1 = \{2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}\}$ and we extrapolate to up to 262k processes. The difference between prediction and measurement never exceeds 7%. Using more training and evaluation data points by adding measurements such that $P_2 = P_1 \cup \{2^{12}, 2^{13}\}$, the model becomes even more precise.

That the requirements models for both the number of bytes and the number of messages received predict almost constant values independent of the number of processes suggests that any increase in communication time is caused by wait states. Because the wavefront travels along the diagonal of the process grid, waiting times proportional to the square root of the number of processes can actually be expected. Having waiting time grow with $\sqrt{p}$ means that every quadrupling of p will double its amount, which can hardly be classified as scalable.

Table 2.2: Models of Sweep3D regions compared on the Juqueen and Juropa systems.

Kernel	JuQueen model	JUROPA model
`sweep → MPI_Recv`	$4.03\sqrt{p}$	$0.51\sqrt{p}$
`sweep`	582.19	70.21
`global_int_sum → MPI_Allreduce`	$1.06\sqrt{p} + 0.03\sqrt{p}\log p$	$0.05\sqrt{p} + 0.01\sqrt{p}\log p$
`sweep →MPI_Send`	$11.49 + 0.09\sqrt{p}\log p$	$0.04 + 2 \cdot 10^{-4}\sqrt{p}\log p$
`source`	$6.8609 + 9.13 \times 10^{-5}\log p$	1.0345

Because the amount of waiting time in Sweep3D, which is responsible for the bulk of the time spent in MPI at larger scales, depends on the progress of the wavefront computation, earlier studies [40, 41] concluded that single-node performance is the most serious impediment to the scalability of Sweep3D—and not, for example, the saturation of network resources. To see whether we arrive at the same conclusion using our automated approach, we also conducted experiments on Juropa, whose cores are much more powerful than Juqueen's. Results for the two platforms obtained with training data from runs with up to 2,048 processes are again consistent with manually developed models. While the sweep() routine, where the wavefront computation takes place, is about eight times faster on Juropa, the receive inside, where the wait states accumulate, is eight times slower on Juqueen. Otherwise, the models we generate for the two kernels on Juqueen and Juropa are identical. Therefore, we can conclude that single-node performance is indeed the most serious impediment to the scalability of Sweep3D.

Table 2.2 contrasts our results for the two platforms with training data sets from runs with up to 2,048 processes. While the coefficients differ, the model hypotheses seem to be largely portable. Coefficients aside, the only differences are a constant term for the allreduce and the $\times 10^{-5}\log p$ term in source(), both of which are inconsequential with respect to performance.

On a final note, this relationship sheds light also on a performance phenomenon observed in a more recent experimental study of Sweep3D [42], which analyzes the consequences of load imbalance between a central rectangular region and the rest of the process grid, which is caused by a corrective function invoked only during certain iterations. Since overload has effects similar to processors with lower speed, it is likely to enlarge only the coefficient of the $\sqrt{p}$ term in the model of the dominant receives and, thus, to have only little bearing on the general scalability. Applying our tool to the affected iterations only, we found this coefficient to be enlarged by 20% but otherwise observed the same scaling behavior.

2.7.2 MILC

In this case study, we show that our tool characterizes also a scalable application correctly. In addition, we show how our tool derives time and requirements models for model parameters beyond the number of processes. MILC is a set of codes written in C for studying quantum chromodynamics (QCD) via parallel simulations of the SU(3) lattice gauge theory on a four-dimensional lattice. In earlier work [1], analytical models were manually created that describe

Table 2.3: Automatically generated models of selected functions in MILC when varying the number of processes. The prediction errors were computed with resepct to a target scale of 65,536 processes.

Kernel	Model [s] $t = f(p)$	$\lvert 1 - R^2 \rvert$	Predictive error [%]
`CGSF`	0.024	0	0.43
`MPI_Allreduce`	$6.30 \cdot 10^{-6} \cdot \log^2 p$	0.084	12.77
`MASFL`	0.0038	0	0.04

the behavior of MILC/su3_rmd, one of the MILC codes, by characterizing its most important components with respect to a number of parameters. We now show that our modeling tool chain allows similar models to be derived automatically.

We first consider weak scaling runs on Juqueen, increasing the number of processes linearly with the problem size. The existing models suggest that MILC is a highly scalable code, that is, the time per process should remain constant except for a rather small logarithmic term caused by global convergence checks. As we show below, our method correctly determines the most important features of this model. Specifically, we demonstrate the tool's ability to derive scalability models for the execution time of three representative kernels: $compute\_gen\_staple\_field$, $g\_vecdoublesum \rightarrow MPI\_Allreduce$, and $mult\_adj\_su3\_fieldlink\_lathwvec$, which we abbreviate as CGSF, MPI_Allreduce, and MASFL, respectively. Given that MILC is known to scale well, we refined the default setting for I by adding $\left\{\frac{1}{3}, \frac{2}{3}\right\}$, as suggested in Section 2.3.3. We collected five data points for each function at the scales $P_3 = \left\{2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}\right\}$ with a local lattice size of $V = 9^4$ per process. All model functions generated for Juqueen are shown in Table 2.3. For ease of understanding, we show $\lvert 1 - R^2 \rvert$, the absolute difference between R^2 and the optimum, which can be considered a normalized error, in the table.

Our models allow a direct performance comparison for the computation kernels CGSF and allreduce. Kernel 2 performs additional communication which scales logarithmically. Our modeling framework identifies a $\log^2 p$ component which is caused by a global reduction for checking convergence. Here, the expected model would be $\log p$, however, experiments on Juqueen have shown that using other communicators than MPI_COMM_WORLD cause performance degradation when using the allreduce collective. MILC uses such communicators, which explains the difference from expectation.

Beyond scalability in terms of the number of processes, we also derive requirements models for the size and number of MPI point-to-point messages as a function of grid points per process. This demonstrates the ability of our tool to generate models for different input parameters—in this example to predict the effects of different process-local grid sizes. For four different performance-critical kernels, the handcrafted model characterizes the message size as $18s\sqrt[4]{V^3}$ bytes, with s being the size of a floating-point value in bytes and V being the number of grid points per process. Our input measurements, which we took

Table 2.4: Automatically generated models of selected functions in MILC when varying the number of grid points per process. For the underlying experiments, we used the following parameters: `meas=5`, `warms=0`, `trajecs=1`, `traj_between_meas=1`, `steps_per_trajectory=10`.

Flops		Invocations		Flops/invocation	
Model	$\lvert 1-R^2\rvert$	Model	$\lvert 1-R^2\rvert$	Model	$\lvert 1-R^2\rvert$
$flops = f(V)$	$[\cdot 10^{-3}]$	$invocations = f(V)$	$[\cdot 10^{-3}]$	$\frac{flops}{invoc.} = f(V)$	$[\cdot 10^{-3}]$
load_lnglinks					
$56,426 \cdot V$	0.03	2,310	0	$24.42 \cdot V$	0.03
load_fatlinks_cpu					
$1,954,230 \cdot V$	0.21	71,430	0	$27.36 \cdot V$	0.21
ks_congrad					
$1.16 \cdot 10^8 + 3.24 \cdot 10^5 \cdot V^{\frac{5}{4}}$	0.292	$5.11 \cdot 10^4 + 13,836 \cdot V^{\frac{1}{4}}$	4	$15.94 \cdot V$	0.143
imp_gauge_force_cpu					
$1,649,790 \cdot V$	0.015	74,040	0	$22.28 \cdot V$	0.015
eo_fermion_force_twoterms_site					
$4,015,930 \cdot V$	0.002	127,050	0	$31.61 \cdot V$	0.002

on Juropa with its more generous memory per node, were made with a fixed number of processes $p = 32$, single precision ($s = 4$ bytes), and a varying number of grid points $\mathcal{V} = \{81, 256, 400, 625, 900, 1080, 1296, 1512, 1764, 2058, 2401\}$. Since there is no performance variation in these requirements measurements, the quality of the automated fit (and thus the confidence) is high, resulting in a model that matches the handcrafted counterpart exactly. Our method also found the number of messages in each kernel to be invariant regardless of the lattice size, which further matches the models in [1]. Another metric analyzed was the number of floating-point operations in each invocation of the time-intensive kernels as a function of the number of grid points per process. The results in Table 2.4 show that the number of floating-point operations per kernel invocation is proportional to the number of grid points (rightmost column), which is again consistent with [1]. All kernels but the conjugate-gradient kernel (`ks_congrad`) have a constant number of invocations, whereas the number of times the conjugate-gradient kernel is invoked depends for this particular input matrix on the number of grid points (middle column).

In summary, our method was able to reproduce the most significant parts of the models that were manually created to describe the behavior of MILC as a function of the number of processes or the local volume. The requirement models for the number of messages, their sizes, and the number of floating-point operations per lattice point can be very useful for architecture co-design. We did not show additional models for cache misses and other metrics because

they follow the same principle. Our results demonstrate that automation can lead to good performance models with low manual effort.

2.7.3 HOMME

To showcase how our tool helps to find hidden scalability bugs in a production code for which no performance model was available, we applied it to HOMME [43], the dynamical core of the Community Atmospheric Model (CAM) being developed at the National Center for Atmospheric Research (NCAR). HOMME, which was designed with scalability in mind, employs spectral element and discontinuous Galerkin methods on a cubed sphere tiled with quadrilateral elements. While experiences in the past did not indicate any scalability issues with up to 100,000 processes, HOMME was never subjected to a systematic scalability study. All the results we present here for this code reflect measurements on Juqueen based on an input configuration suggested by the application developer team.

Table 2.5 lists different kernels of the code, ordered by their asymptotic runtime ($p_t \to \infty$). It shows the models produced for two different sets of input configurations. The first one includes data points at the scales $P_4 = \{600, 1176, 4056, 7776, 13824, 14406, 15000\}$, the second one $P_5 = P_4 \cup \{15606, 16224, 23814, 31974, 43350\}$ adds more measurements to the initial set. The order in the table is based on models determined using P_5. The models derived from P_4 show constant runtimes for all kernels except for the reduce in $box\_rearrange$, which grows with p^3. Deriving the models from the larger set introduces a dependence on p^2 (with a small factor) for all but one of the hitherto constant kernels. Obviously, the enlarged set reveals a phenomenon not visible in the smaller set. If the number of processes is large enough, both the quadratic and the cubic terms will turn into serious bottlenecks, contradicting our initial expectation the code would scale well. The table also shows the predictive error, which characterizes the deviation of the prediction from measurement at the target scale $p_t = 130k$, highlighting the benefits of including the extra data points.

After looking at the number of times any of the quadratic kernels was invoked at runtime, a metric we also measure and model, the quadratic growth was found to be the consequence of an increasing number of iterations inside a particular subroutine. Interestingly, the formula by which the number of iterations is computed contained a *ceiling* term that limits the number of iterations to one for up to and including 15k processes. Beyond this threshold, a term depending quadratically on the process count causes the number of iterations executed to grow rapidly, causing a significant drop in performance. It turned out that the developers were aware of this issue and had already developed a temporary solution, involving manual adjustments of their production code configurations. Specifically, they fix the number of iterations and carefully tune other configuration parameters to ensure numerical stability. Nevertheless, the issue was correctly detected by our tool. Given the tuning necessary to ensure numerical stability, a weak scaling analysis of the workaround is beyond the scope of this paper.

In contrast to the previous problem, the cubic growth of the time spent in the reduce function was previously unknown. The reduction is needed to funnel data to dedicated I/O processes. The coefficient of the dominant term at scale is very small (i.e., in the order of 10^{-13}). While not

Table 2.5: Models of the kernels of HOMME derived from smaller and larger-scale input configurations. The predictive error refers to the target scale of $p_t = 130k$.

$P_4(p_i \leq 15,000)$		$P_5(p_i \leq 43,350)$	
Model [s] $t = f(p)$	Predictive error [%]	Model [s] $t = f(p)$	Predictive error [%]
`box_rearrange` → `MPI_Reduce`			
$2.53 \cdot 10^{-6} \cdot p^{1.5} + 1.24 \cdot 10^{-12} \cdot p^3$	57.02	$3.63 \cdot 10^{-6} \cdot p^{1.5} + 7.21 \cdot 10^{-13} \cdot p^3$	30.34
`vlaplace_sphere_wk`			
49.53	99.32	$24.44 + 2.26 \cdot 10^{-7} \cdot p^2$	4.28
`laplace_sphere_wk`			
44.08	99.32	$21.84 + 1.96 \cdot 10^{-7} \cdot p^2$	2.34
`biharmonic_wk`			
34.40	99.33	$17.92 + 1.57 \cdot 10^{-7} \cdot p^2$	3.43
`divergence_sphere_wk`			
16.88	99.31	$8.02 + 7.56 \cdot 10^{-8} \cdot p^2$	4.25
`vorticity_sphere`			
9.74	99.55	$6.51 + 7.09 \cdot 10^{-8} \cdot p^2$	8.66
`divergence_sphere`			
15.36	99.33	$7.74 + 6.91 \cdot 10^{-8} \cdot p^2$	0.95
`gradient_sphere`			
14.77	99.33	$6.33 + 6.88 \cdot 10^{-8} \cdot p^2$	5.17
`advance_hypervis`			
9.76	99.25	$5.5 + 3.91 \cdot 10^{-8} \cdot p^2$	1.47
`compute_and_apply_rhs`			
48.68	1.65	49.09	0.83
`euler_step`			
28.08	0.51	28.13	0.33

being visible at smaller scales, it will have an explosive effect on performance at larger scales, becoming significant even if executed just once. Due to noise in the data the value of such small coefficients might not be exact, leading to errors in the prediction, but accuracy is secondary to locating scalability bottlenecks. The reason why this phenomenon remained unnoticed until today is that it belongs to the initialization phase of the code that was not assumed to be performance relevant in larger production runs. While still not yet crippling in terms of the overall runtime, which is in the order of days for production runs, the issue already cost more than one hour in the large-scale experiments we conducted. The problems was reported to the developers at NCAR, who are currently working on a solution. The example demonstrates the advantage of modeling the entire application instead of only selected candidate kernels

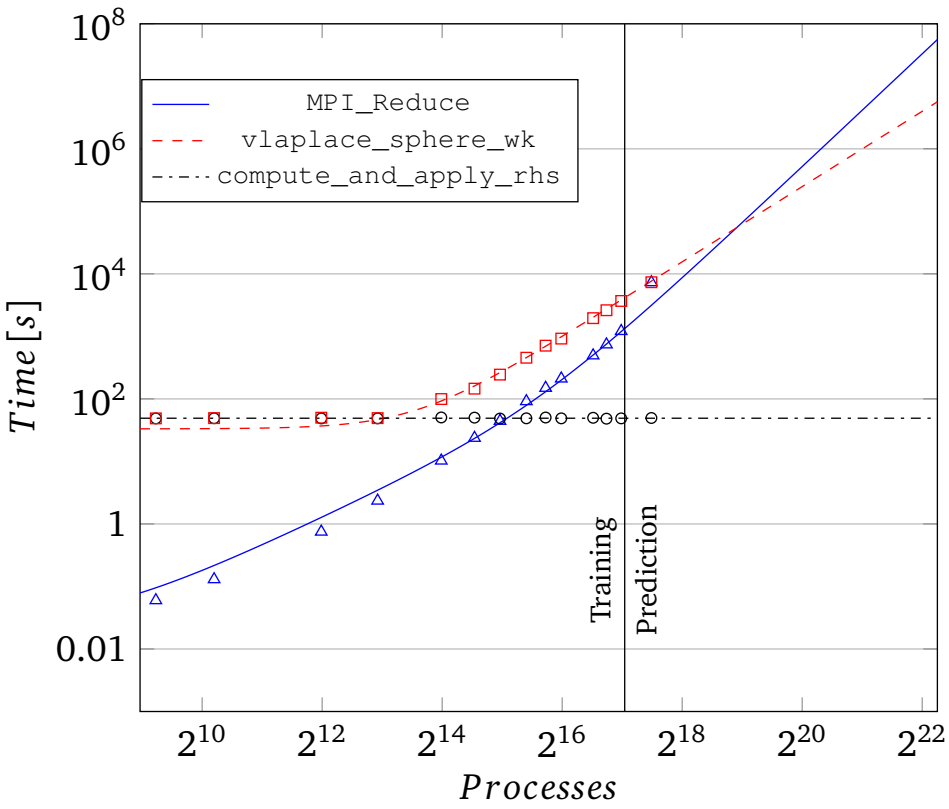


Figure 2.5: Runtime of selected kernels in HOMME as a function of the number of processes. The graph compares predictions (dashed or contiguous lines) to measurements (small triangles, squares, and circles).

expected to be time intensive. Some problems might simply escape attention because non-linear relationships make our intuition less reliable at larger scales. Note that coefficients such as 10^{-13} are small in view of the typical run-to-run deviation, but have to be seen in relation to the associated polynomial expression p^3, which became larger than 10^{12} in our input experiments. Given that the target scale is usually one or more orders of magnitude greater than the largest input scale, fully accurate coefficients are therefore secondary when trying to locate scalability bottlenecks with higher exponents of p.

Figure 2.5 summarizes our two findings and compares our predictions with actual measurements. While the quadratically growing iteration count seems to be more urgent now, the reduce might become the more serious issue in the future.

2.7.4 UG4

UG4 is a simulation framework [7] capable of solving partial differential equations. It uses grid based discretization methods such as the finite element method or the vertex-centered finite volume method. Complex physical geometries are resolved by hybrid, unstructured, adaptive, hierarchical grids in up to three space dimensions. The strong focus the developers have on efficient and highly scalable solvers, both when using algebraic and geometric multigrid methods have lead to a joint study with Goethe University Frankfurt [44].

Table 2.6: Models for CG solver kernels of UG4 in a weak scaling study[7]. Models are provided for the execution time and the number of times each solver kernel is invoked.

Time		Invocations	
Model	$\lvert 1-R^2\rvert$	Model	$\lvert 1-R^2\rvert$
time $= f(p)$ [ms]	$[\cdot 10^{-3}]$	invocations $= f(p)$	$[\cdot 10^{-3}]$
CG → norm			
$3.74 + 4.65 \cdot \sqrt{p}$	0.764	$75.6 + 117.7 \cdot \sqrt{p}$	0.102
CG → dotprod			
$8.83 + 13.3 \cdot \sqrt{p}$	0.475	$149.2 + 235.4 \cdot \sqrt{p}$	0.102
CG → SparseMatrix_axpy			
$96.3 \cdot \sqrt{p}$	0.398	$75.6 + 117.7 \cdot \sqrt{p}$	0.102
CG → VecScaleAdd			
$13.7 + 22.3 \cdot \sqrt{p}$	0.088	$222.9 + 353.1 \cdot \sqrt{p}$	0.102

Extra-P was used to generate performance models for a number of configurations relevant to the application developers and confirm the detailed performance expectation the developers had of their implementation. The studies were focused on use cases of particular interest to developers due to the frequency they are needed and how much computation effort they require.

The first study focused on the weak scaling behavior of the conjugate gradient solver in UG4. The models, represented in Table 2.6, show a $\sqrt{p}$ growth, which is expected and explained by the increase in grid refinement due to the larger problem size. Whilst this is a scalability issue, it is not a performance bug. Given how conjugate gradient solvers work, the growth is the result of the increase of work per process inherent in increasing the problem size. It is not an implementation issue in UG4. This is reflected by the increase in number of invocations, which has the same growth rate as the execution time itself.

The second study was a weak scaling analysis of a three dimensional skin model simulation using UG4. We analyzed the time needed by network communication collectives and also the bytes send over the network. Furthermore we investigated the runtime of relevant computational kernels. The study confirmed the developer expectations and proved that the solver scales very well. A summary of the resulting models is included in Table 2.7.

We note that the performance modeling process was significantly simplified in the case of UG4 by the expectations the developers already had for the performance of this application. This made the validation and verification of the empirically generated models fast and straightforward as all behaviors were easily explained by already existing assumptions.

Table 2.7: Weak scaling study of the three-dimensional skin model simulation. Models for kernels creating MPI communicator groups are displayed in the top table. Models for sparse matrix assembling ($assemble\_linear$) and multigrid ($GMG->*$) are found in the bottom table[7].

Time		Bytes sent	
Model	$\lvert 1-R^2\rvert$	Model	$\lvert 1-R^2\rvert$
time $= f(p)$ [ms]	$[\cdot 10^{-3}]$	bytes $= f(p)$	$[\cdot 10^{-3}]$
`LoadUGScript` → `MPI_Allreduce`			
$9.33 + 0.91 \cdot \log p$	42.6	$4 \cdot \mathcal{O}($`MPI_Allreduce`$)$	0.000
`init_levels` → `MPI_Allreduce`			
$27.3 + 1.3 \cdot \log p^2$	19.6	$80.03 \cdot p \cdot \mathcal{O}($`MPI_Allreduce`$)$	0.003
`init_top_surface` → `MPI_Allreduce`			
$3.71 + 5.18 \cdot p^{1/4}$	9.88	$4 \cdot p \cdot \mathcal{O}($`MPI_Allreduce`$)$	0.000

Time		Invocations	
Model	$\lvert 1-R^2\rvert$	Model	$\lvert 1-R^2\rvert$
time $= f(p)$	$[\cdot 10^{-3}]$	invocations $= f(p)$	$[\cdot 10^{-3}]$
`GMG` → `PreSmooth` → `jacobi`			
$1.89 \cdot 10^{-2} + 0.04 \cdot 10^{-2} \cdot \log p$	42.6	$70.6 + 1.4 \cdot \log p$	76.9
`GMG` → `prolongate`			
$4.24 \cdot 10^{-2} + 0.10 \cdot 10^{-2} \cdot \log p$	84.4	$23.5 + 0.5 \cdot \log p$	76.9
`assemble_linear`			
1.68	0	1	0

2.7.5 XNS

XNS [2] is a finite element flow simulation code. XNS supports a wide variety of flow configurations in both two- and three-dimensional domains and is actively used in numerous scientific projects, such as designing an efficient blood pump, a simulation of which is shown in Figure 2.7.

Together with the developers we performed an analysis of the strong scaling performance of XNS with Extra-P. Unlike the weak scaling studies previously presented, we do not model the average execution time measured across all processes but rather sum up all measurements across all processes. Ideally, we would expect the total runtime to remain constant as we increase the number of processes. Of course, some functions, such as any MPI communication calls, will have an increased runtime. However, as a rule of thumb, even communication operations

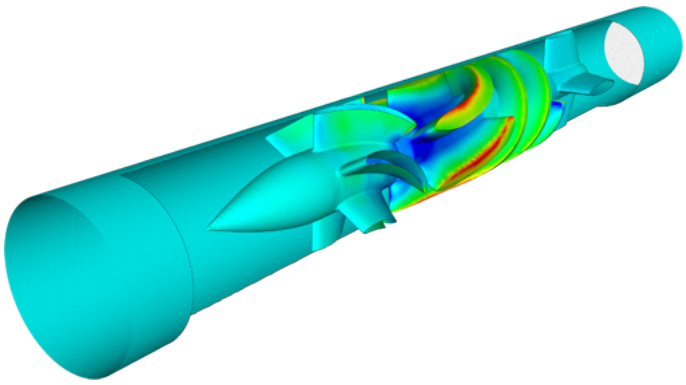


Figure 2.6: Graphical representation of blood flowing through a blood pump. The simulation was performed with XNS [2].

should not scale worse than linearly. For example, if every process has to receive a byte at the start of execution and send another byte at the end, we would expect that the total runtime spent sending and receiving messages to grow linearly with the number of processes, as more and more processes perform the two operations.

We were able to discover a scalability bug in an initialization routine which caused the parallel communication to grow quadratically rather than linearly as more processes were used. This lead to the MPI receive operation in question to take up more than half the runtime of XNS on 4096 processes when it only took less 0.5% of the runtime when only 128 processes where used, as shown in Table 2.8. The developers have fixed the issue and our analysis did not uncover further scalability bugs. The *ewddot* kernel contains most of the computation done by XNS. Whilst a small inefficiency was also detected there, it is a much smaller issue that the previous performance bug identified. Not only is the growth rate of the *ewddot* kernel smaller than linear with the number of processes, the contribution of the growing part is significantly smaller than the constant term of the model. For current production runs, improving the efficiency of this kernel was not considered a priority by the developers. The overall execution time of these kernels is displayed in Figure 2.7.

Discovering the performance issue of the MPI receive operation in the oft overlooked initialization phase of XNS reinforces the need to provide full coverage of the code when analyzing the performance of an application.

Table 2.8: Models for two selected XNS kernels (i.e., call paths) The values and models reflect exclusive execution times without callees. The percentage of the total execution time taken by the respective kernels are compared between $p = 128$ and $p = 4.096$

Runtime [%] at $p = 128$	Runtime [%] at $p = 4.096$	Model [s] for $t = f(p)$
`ewdgennprm → MPI_Recv`		
0.46	51.46	$0.029 \cdot p^2$
`ewddot`		
44.78	5.04	$37406.80 + 13.29 \cdot \sqrt{p} \cdot \log p$

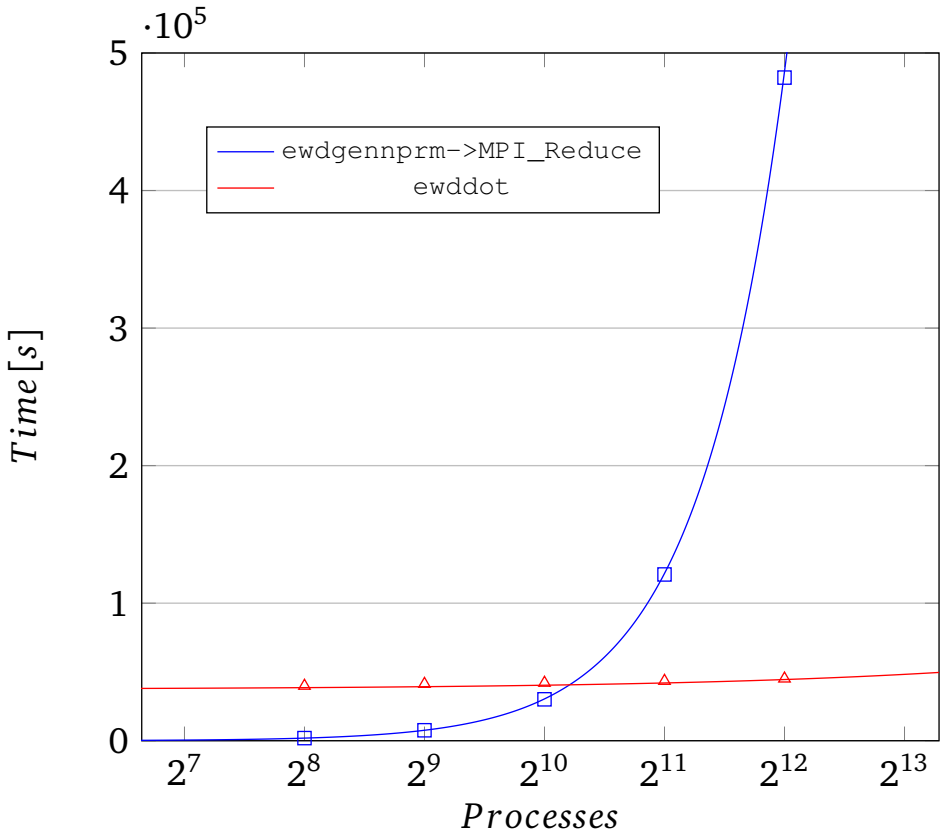


Figure 2.7: Runtime of selected kernels in XNS as a function of the number of processes. The graph shows models (contiguous lines) and measurements (small triangles and squares).

2.8 Discussion

In this chapter we have introduced a new approach to determine performance models empirically using measurements of as few as five different parameter configurations. We have validated our method with both synthetic tests and scientific applications for which analytical models are available. We further used Extra-P to model the behavior of applications for which analytical models did not previously exist, and were able to uncover previously unknown scalability bugs. Using the technique described here as foundation and stepping stone, we will expand its capabilities to tackle more complex performance questions, from quantifying the effect of multiple parameters simultaneously to helping co-design future HPC systems through performance model-aided requirements engineering.

3 Multi-Parameter Performance Modeling

The second contribution of this work is an approach that allows the effect of multiple performance relevant parameters to be modeled at the same time. This allows for the discovery of compounded performance effects.

3.1 Multi-parameter modeling

Common questions asked by developers when trying to understand the behavior of applications are:

- How does application performance change when more processors are used?
- How does application performance change when the problem size is increased or decreased?

Changing the processor count while keeping everything else fixed is also known as strong scaling. The goal of many large-scale applications, however, is to solve larger problems using more processing power, leading to the concept of weak scaling. Weak scaling is often defined as the application's behavior when the problem size *per processor* is fixed and the processor count is varied. Creating an experimental setup in practice where the problem size per processor is fixed is not trivial, as the problem decomposition is not arbitrary in the general case. When considering the pressure on applications to judiciously use computing resources both questions must be answered, and a new vital question arises:

- Are the effects of processor variation and problem size variation independent of each other or can they amplify each other?

For example, a weak-scaling run of the kernel `SweepSolver` in Kripke [45], a particle transport proxy application, has a runtime model for processor variation of $t(p) = O(p^{1/3})$ and a runtime model for varying the number of dimensions of $t(d) = O(d)$. The number of dimensions influences the problem size proportionally. It now needs to be determined how these two factors interact. Depending on their interaction, the application is scalable or not. For example, it would make a huge difference if the combined effect of processor variation and number of dimensions was $t(p,d) = O(p^{1/3} \cdot d)$ or $t(p,d) = O(p^{1/3} + d)$.

Furthermore, the problem size can often be decomposed into multiple independent parameters, such as length, width and depth for an applications describing a three dimensional space. Understanding the effect of each individual parameter on performance can help developers pinpoint implementation inefficiencies. Another advantage is the potential to highlight how choices in the implementation of the algorithm can lead to theoretically equally relevant parameters having different effects on performance.

3.1.1 A normal form for multiple parameters

Below, we expand the original performance model normal form presented in Section 2.1 to include multiple parameters.

$$f(x_1,\ldots,x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \tag{3.1}$$

This expanded normal form allows a number m of parameters to be combined in each of the n terms that are summed up to form the model. Each term allows each parameter x_l to be represented through a combination of monomials and logarithms. The sets $I, J \subset \mathbb{Q}$ from which the exponents i_{k_l} and j_{k_l}, respectively, are chosen can be defined as in the one-parameter case.

Our experience with single-parameter modeling leads us to the conclusion that with as few as five different input values only one term can be meaningfully used without overfitting the data. However, enough terms have to be allowed such that the effect of each parameter can be modeled independently. This leads us to the following rule of thumb regarding number of terms: $n = m$, unless significantly more data points are available for each parameter. In practice, we use $n = m'$, where m' is the number of parameters whose effect on performance is not constant across all gathered measurements.

3.1.2 Challenges for multiple parameters

Of course, if multiple parameters are considered, performance experiments have to be conducted for all combinations of parameter values and the total number of experiments that is required grows accordingly. While this might be manageable if the number of parameters considered is small enough (single digit) and/or the cost of an individual experiment is very small, another and more serious problem emerges even for two and three parameters.

Looking at Equation 3.1, the combinatorial explosion of the search space for model hypotheses that multi-parameter modeling generates becomes apparent. This shows the need for efficient methods for traversing the search space. The maximum number of terms should always allow at least an additive combination of all parameters so $n \geq m$. For convenience, we will use $n = 3$ throughout the following example, which is required for three parameters. With $n = 3$ and I, J defined as in Section 2.1, the model search space when one parameter is considered can be built as follows: For each of the terms there are $|I| \cdot |J|$ possible options, meaning 54 possibilities in this case. The order of terms is irrelevant and the same term cannot be repeated, therefore the cardinality of the search space is the binomial coefficient $\binom{|I|\cdot|J|}{n}$, i.e., 24,804 models in total. If we now consider two parameters, each term has $(|I| \cdot |J|)^2$ possible options, i.e., 2,916 in total. The model search space in this case contains 4,128,234,660 candidates. Let us assume that 300,000 hypotheses can be evaluated in a second, a rate drawn from our experience on current commodity personal computers. Even with the simplifying assumption that evaluating a hypothesis with multiple parameters would take as much as evaluating a hypothesis with only one parameter, it would still take more than three hours to select the best model for a single combination of metric and kernel. With three parameters the model search space contains

around $6.51 \cdot 10^{14}$ candidates, and with m parameters $\binom{(|I|\cdot|J|)^m}{n}$, making the search for the best fit a daunting task. Spending six years to compute the best model with three parameters for one metric of one kernel is not something any developer would consider. Obviously, one does not need many parameters to make the traversal of such a multi-parameter search space practically infeasible. While this problem is embarrassingly parallel, the resource requirements for such a performance modeling process will far outweigh any gains obtained through optimization of the target application. To overcome the challenge that the size of the search space presents, we speed up the search process using novel heuristics, which we describe in Section 3.2.

3.2 Fast multi-parameter modeling

We build on the concepts of single-parameter modeling from our prior work, but extend and optimize them to match the new requirements posed by modeling multiple arbitrary parameters. Note that the original method is only capable of modeling the scalability as a function of one parameter, usually the number of processes. This single-parameter modeling rests on several simplifying assumptions that do not hold for general multi-parameter modeling, for example, that we can search all models for this one parameter. Thus, we first develop a new search method for the model space of a single arbitrary parameter and then derive an effective method to combine all single-parameter models into a single model for all parameters.

3.2.1 Improved single-parameter modeling

To model multiple parameters without the time to solution becoming prohibitive, with billions of candidates being generated for as few as three parameters, the existing approach to model detection is no longer sufficient: we must find a way to reduce the search space of model hypotheses. The following method is only applicable to single-parameter modeling. It complements the hierarchical search outlined in the next subsection to speed up the entire modeling process, as the hierarchical search heuristic uses the single-parameter modeling as a starting point.

Reducing a search space is often related to finding some ordering of the search space, i.e., finding a way to rank the possible hypotheses, and our method is no exception. We use the following sets of modeling terms for $n = 1$: $I = \left\{\frac{0}{4}, \frac{1}{4}, \ldots, \frac{12}{4}\right\}$, and $J = \{0\}$ as an example to demonstrate our hypothesis ranking approach. This example generates a small set of hypotheses when modeling a single parameter x: $\{x^0; x^{1/4}, \ldots, x^{12/4}\}$. Based on our experience with performance modeling, we make the following observation: if we rank the hypothesis functions by the magnitude of their first derivative at the observation with the largest parameter value then the respective error function of the ranked hypotheses is unimodal, i.e., a function that has a unique minimum or maximum point and is monotonically decreasing/increasing towards it. The unique minimum of this discrete function will be the best matching term.

This is intuitive: the regression and cross-validation approach we use always finds the best possible coefficients for each of the terms to fit the available data, but the results will be better, the higher the similarity between the hypothesis function and the true function is. As an exam-

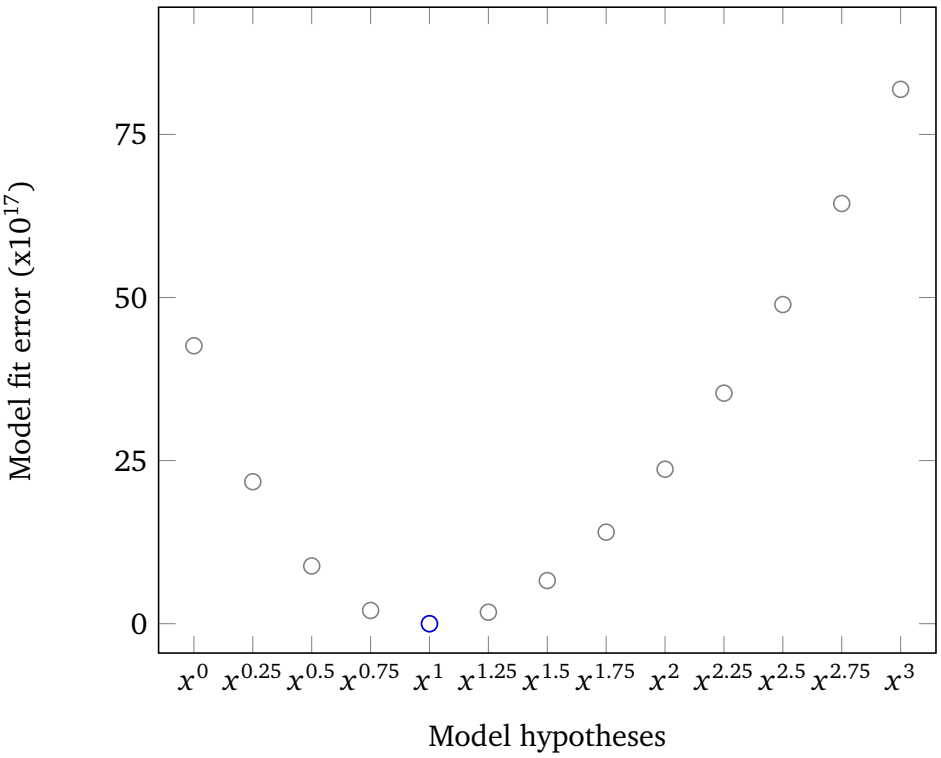


Figure 3.1: Model fit error for different model hypotheses. The fit error of model hypotheses decreases towards the one with the smallest error, in this case x^1.

ple, let us assume the true function is $f_t(x) = 2 \cdot x^2$. If we have two hypotheses, $f_1(x) = c_1 \cdot x$ and $f_2(x) = c_2 \cdot x$, the regression method we apply will find c_1 and c_2 such that both f_1 and f_2 are as close as possible to f_t (the error of the fit is as small as possible). However, as the growth of the linear function f_1 is much closer to the quadratic function f_t than the growth of the logarithmic function f_2, and therefore the error of the linear function f_1 will be smaller than the one of the logarithmic function f_2.

Following this observation, we sort all hypotheses by their slopes at the measurement with the largest parameter value. In the particular example from above, this is trivial, as all hypotheses are simple monomials and thus their order (for any value) is the ascending order of the exponent. In the case of more complex hypotheses the order may change depending on the measurement chosen. For example comparing combinations of polynomials and logarithms with different exponents can lead to intervals where one combination or another have a higher growth rate, depending on the parameter range. As a way of imposing a deterministic total ordering, we select the measurement with the largest parameter value, as users are most often interested in understanding and predicting the behavior at and beyond the upper range of a given parameter. As an example, we will attempt to model the effect that varying the group number has on floating-point instructions executed in the `LTimes` kernel of the Kripke application. We consider the following pairs of parameter x and measurement t as in input: $(32, 1209.6)$, $(64, 2419.2)$, $(96, 3628.8)$, $(128, 4838.4)$, $(160, 6048)$. Figure 3.1 shows the residual sum of squares error of various model functions fitted via the least squares method to our five data points. The x^1 hypothesis is a perfect fit without error, and the error of the

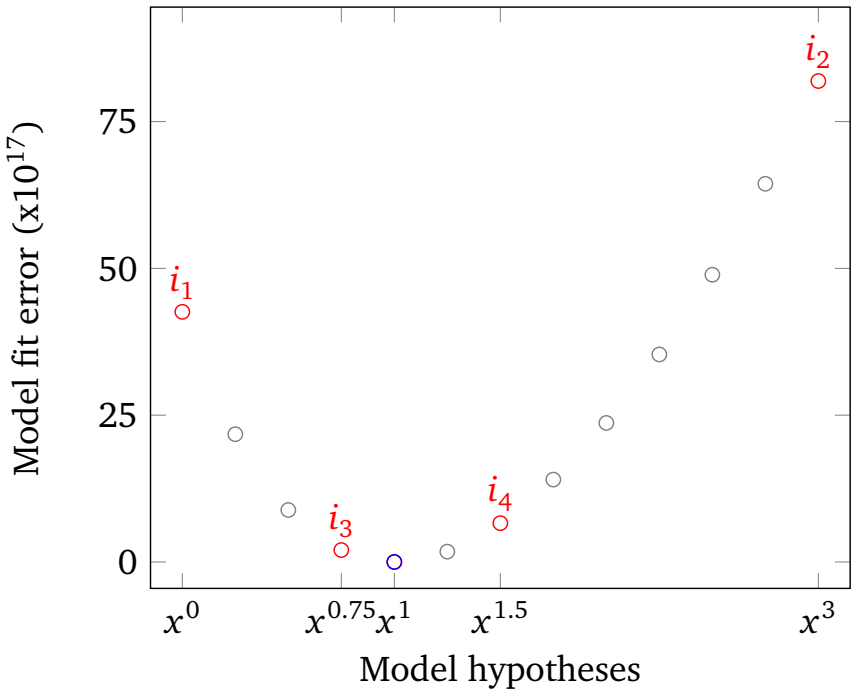


Figure 3.2: Golden section search interval reduction. The search interval starts as $[i_1, i_2]$ and becomes $[i_1, i_4]$ after one step of the golden section search method.

model fit as a function of its rank (the index in the sorted hypothesis list) has a minimum at the index of the best model. Since this minimum is also unique, this function is unimodal in the analyzed range.

3.2.1.1 Modified golden section search

The observation above justifies a modified golden section search as a means to traverse the model hypothesis search space. This method is a way to quickly narrow down the range of values in which the extremum of a unimodal function is found. Starting with the complete search space, we recursively refine the interval in which the extremum can be found as follows: we first divide the total search space into subintervals by choosing two additional points between the extreme points of the interval. For optimal performance, the points in the interval are selected using the golden ratio $\phi = 1.618$. We then evaluate the model fit at all four points and from there pinpoint the interval that contains the extremum. We then repeat the same approach on this interval recursively, until only one hypothesis remains.

As an example, a step of the method is displayed in Figure 3.2 using data from the `LTimes` kernel of Kripke: the two end points of the interval $[i_1, i_2]$ and one point in the interval, i_3, such that $\frac{i_2 - i_3}{i_3 - i_1} = \phi$. In this situation $e(i_1) > e(i_3) < e(i_2) \wedge e(i_1) > e(i_2)$. A new point i_4 is then chosen in the interval $[i_3, i_2]$ using the same ϕ as before. Indices only take integer values, so the i_3 and i_4 must be rounded before the hypothesis error function e can be evaluated.

The evaluation of i_4 indicates where the search should continue. If $e(i_4) \geq (i_3)$ due to the monotonicity of e, the minimum cannot be in the interval $[i_4, i_2]$. Therefore the search has to be continued in the interval $[i_1, i_4]$. Should $e(i_4) < e(i_3)$ the search must be continued in

$[i_3, i_2]$. After a finite number of recursive search interval contractions only one hypothesis can be selected, and that will be the one with the optimal fit out of all hypotheses available in the search space.

3.2.1.2 Limitations

If the true function we are trying to model has a behavior very different from what can be modeled based on the normal form then it is possible that the above observation no longer holds. Examples include discontinuous functions and functions with multiple behaviors depending on the parameter values. If they occur, in the worst case a model which is not the model with the best fit could be selected. Nevertheless, if a model has a large fit error, an unsatisfying value for $\hat{R}^2$ would alert the user before he could draw any wrong conclusions. A new approach that is capable of handling these corner cases has been developed and will be described in more detail in Section 5.5.

3.2.1.3 Benefits

Golden section search allows the model hypothesis space to be searched faster. The dependence between the cardinality of the hypotheses set and the number of steps needed to find the best model goes down from linear to logarithmic. The benefits therefore increase the larger the search space becomes. For example, in the case of the single parameter search described in Section 3.1.2, which created a search space of 24,804 candidates, the number of steps required drops to 25, a reduction of almost three orders of magnitude, as in each step at least a third of the candidates are discarded.

The advantage of the golden section search over similar approaches, such as ternary search, is the reuse of previous measurements. At any given step only one new point has to be evaluated. Needing as few such evaluations as possible is crucial, as this is a computationally intensive part of the process.

3.2.2 Combining multiple parameters

Our approach for multi-parameter modeling is based on the assumption that the best single-parameter models for each individual parameter form the best multi-parameter model together, only their combination is unknown. This is—just like the previous assumption—intuitive: If the best model for the process count is $c_1 \cdot \log x_1$ and the best model for problem size is $c_2 \cdot {x_2}^2$ we expect that the best multi-parameter model will either be $c_3 \cdot \log x_1 \cdot {x_2}^2$ or $c_4 \cdot \log x_1 + c_5 \cdot {x_2}^2$ depending on whether the effects of the two parameters are combined or independent of each other. We do not expect it to be $c_6 \cdot {x_1}^3 \cdot \sqrt{x_2}$, or any other model unrelated to the best single-parameter models.

3.2.2.1 Hierarchical search

Using the assumption above, we first obtain single-parameter models for each individual parameter using the golden section search method previously described. Once we have these models, all that is left is to compare all additive and multiplicative options of combining said models into one multi-parameter model and choosing the one with the best fit.

The size of the search space for this approach is as follows: given m parameters and one n-term model for each of them. We must combine all subsets of terms of each single-parameter model with each subset of terms of each other single-parameter model. The number of subsets of a set of n elements is 2^n, so the total size of the search space is $2^{n \cdot m}$.

Again using the example from Section 3.1.2, assuming the single-parameter models for all three parameters have been computed and that all models have three terms each (the worst case scenario for search space cardinality in this case), the number of hypotheses that have to be tested is $2^{3*3} = 512$. Adding the 3 times 25 steps needed to generate the single-parameter models, we need to look at most at 587 models to find the best fit, compared to the $6.51 \cdot 10^{14}$ in the unoptimized approach.

3.2.2.2 Discussion

The total size of the search space, $2^{n \cdot m}$, can seem daunting at first, but is insignificant when compared to the unoptimized search space. Considering that two terms have proven sufficient to successfully model applications, the search space becomes 4^m. While the number of parameters cannot be arbitrarily large, our approach practically removes model generation as a bottleneck, as collecting sufficient experimental measurements will prove impractical long before our modeling approach will experience any issues.

3.2.3 Data collection

To create multi-parameter models, we need to have sufficient input data that allows accurate single-parameter models for all parameters to be generated, as required by the hierarchical search described in Section 3.2.2.1. For this reason, the set of parameter assignments used in experiments must be symmetric. Assume each of v measurements is a tuple of $(x_{1,i}, \ldots, x_{m,i}, t_i)$, consisting of m input parameter values plus the metric t of interest (e.g., the completion time). Then symmetric means that for each input parameter x_i there must be a set of k measurements where x_i is varied while all other parameters remain constant. For example, the three-parameter tuple set ((1,10,22),t_1), ((2,10,22),t_2), ((1,11,22),t_3), ((2,11,22),t_4), ((1,10,44),t_5), ((2,10,44),t_6), ((1,11,44),t_7), ((2,11,44),t_8) is symmetric, as all combinations of values are present. The removal of any single tuple would render the set non-symmetric. Symmetry of the measurements used is required by our method because it allows us to fix any single parameter and look at it in isolation, considering the other parameters constant. For this, we project out all but one parameter and calculate the average value across all tuples with the same assignment for the chosen parameter. For example, if we model the first parameter of the

previous example, we would use $((1,0,0),(t_1+t_3+t_5+t_7)/4)$ and $((2,0,0),(t_2+t_4+t_6+t_8)/4)$ as the basis for the single-parameter model of the first parameter. Overall, this strategy requires a full factorial design of $v = k^m$ measurements if each parameter is tested in k configurations. We empirically observed that $k = 5$ is sufficient in practice. Thus, the number of parameter assignments to be tested is 5^m. Depending on the run-to-run variation, the measurement of each parameter assignment must be repeated up to five times. Therefore, the total number of required measurements is between 5^m and $5^{(m+1)}$.

3.3 Evaluation with synthetic data

To evaluate the heuristics presented in Section 3.2, we quantify the size of the search space traversed during the model search in comparison to an exhaustive traversal of the same search space. Furthermore, we determine the frequency at which our heuristics lead to models that differ from the ones the exhaustive search produces. In those cases where the models we discover are different, we analyze these differences and discuss their impact on the quality of the results. Because traversing the entire search space for three or more parameters is prohibitively time consuming even with a very small number of potential terms, we allow only at most two model parameters for the purpose of this comparison.

The evaluation is divided into two parts. First, we examine how close the models, generated both through exhaustive search and with the help of heuristics, are to inputs derived from synthetically generated functions. This allows our results to be compared with a known optimal model. Second, we compare the results of both approaches when applied to actual performance measurements of scientific codes, which factors in the effects of run-to-run variation.

We generate 100,000 test functions by instantiating our normal form from Eq.2.1 with random coefficients $c_l \in (0,100)$ and i_l and j_l randomly selected from the sets I and J, obtaining functions of the type represented in Eq. 3.2.

$$\begin{aligned} f(x) = c_0 + c_1(x^i \cdot log_2^j(x))^{0|1} \cdot (y^k \cdot log_2^l(y))^{0|1} + \\ c_2(\cdot x^i \cdot log_2^j(x))^{0|1} \cdot (y^k \cdot log_2^l(y))^{0|1} \end{aligned} \tag{3.2}$$

To create the input sets for our model generator, we evaluated the functions at 5^m points with $m = 2$ being the number of parameters. To these inputs, our model generator responded in three different ways:

1. **Optimal models.** The most common result (approx. 95%) is that the heuristically determined model, the model determined through an exhaustive search, and the known optimal model are identical.

2. **Lead-order term and its coefficient identified, smaller term not modeled by either method.** Another scenario is encountered when the optimal model has the form $c_1 \cdot f(x) \cdot f(y) + c_2 \cdot f(y)$, where $c_1 \cdot f(x) \cdot f(y) \gg c_2 \cdot f(y)$ in the considered parameter ranges. The optimal model $100 \cdot x^3 \cdot log_2(y) + 2 \cdot log_2(y)$ is an example of this case. Neither modeling approach is capable of detecting the smaller term and they both only model the lead-order term. The effect on the quality of the resulting models is very small, and an attempt to model such small influences will often lead to noise being modeled instead.

Table 3.1: Evaluation of heuristics using synthetic functions.

Search type	Heuristic	Exhaustive
Optimal models identified	95,480 [95.5%]	96,120 [96.1%]
Lead-order term identified (including coefficient)	4,520 [4.5%]	3,880 [3.9%]
Lead-order term not identified	0 [0%]	0 [0%]
Modeling time	1.5 hrs.	107 hrs.

3. **Lead-order term and its coefficient identified, smaller additive term only modeled by exhaustive search.** This behavior appears when the optimal model has the form $c_1 \cdot f(x) + c_2 \cdot f(y)$, where $c_1 \cdot f(x) \gg c_2 \cdot f(y)$. In this case the heuristic approach fails to identify the parameter with a smaller effect. The contribution of one parameter leads to the single-parameter model for the other parameter to have a very large constant component. As this constant is larger than the variation caused by the parameter with the smaller effect, the modeling process attributes the variation to potential noise and conservatively selects the constant model. The effect on the quality of the resulting model is again negligible.

We have not added synthetic noise to this evaluation as it was important to compare the speed of the two approaches as well as their accuracy. Noise would have degraded the performance of both approaches, without offering new insights.

Table 3.1 displays the number of times the modeling identified the entire function correctly and the times only the lead-order term was identified correctly. The lead-order term was correctly identified in all test cases. The difference in time required to obtain the 100,000 models is significant: 1.5 hours when using the heuristics compared to 107 hours when trying out all models.

3.4 Case studies

In addition to synthetic data, we evaluate our heuristics with three scientific applications: Kripke, CloverLeaf, and BLAST. Below, we briefly describe them along with the input decks used. All tests we report were run on Vulcan, an IBM BG/Q system at Lawrence Livermore National Laboratory with 24,576 nodes in 24 racks. Each node is powered by an IBM PowerPC A2 processor with 16 cores/64 hardware threads and features 16 GB of main memory. The system uses IBM's Compute Node Kernel (CNK), as well as an IBM MPI implementation based on MPICH2. We use Score-P [13] to acquire all of our metrics for the chosen applications. In particular, we measure execution time, total number of instructions, number of floating point instructions, and MPI bytes sent and received.

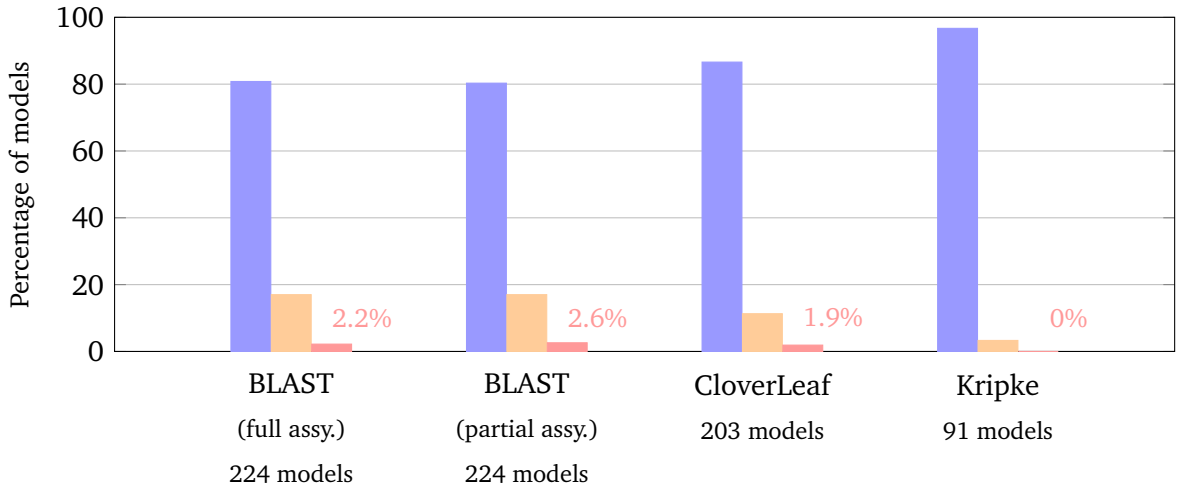


Figure 3.3: Comparison of performance models obtained for scientific applications using either our heuristics or a full traversal of the search space. For each application, we show the percentage of times where the resulting models were identical (left bar), where only the lead-order terms and their coefficients were the same (center bar), and where the lead-order terms were also different (right bar).

3.4.1 Kripke

Kripke [45] is an open-source 3D Sn deterministic particle transport code. It calculates angular fluxes and stores them in a flexible hierarchy of data structures (direction sets, group sets, and zones). Kripke was designed as a research tool to explore how data-layouts affect performance, especially on different architectures and with different programming models. For this test, we varied two parameters: the number of directions per set (16, 32, 64, 128, 256, and 512), and the number of groups per set (32, 64, 96, 128, and 160).

3.4.2 Blast

BLAST [46] is an arbitrary-order finite-element hydrodynamics research code under development at Lawrence Livermore National Laboratory. It is used to explore the costs and benefits of high-order finite element methods for compressible hydrodynamics problems on modern and emerging architectures. BLAST implements two different algorithmic approaches that produce the same answer: *full assembly*, which assembles and solves a global matrix, and *partial assembly*, which stores only physics data and solves the linear system matrix free. Using the Sedov test problem [47] as input, we kept the number of degrees of freedom in the problem fixed and ran 2D tests varying two parameters: order (1, 2, 4, 8, and 16) and number of MPI ranks (64, 256, 1024, 4096, and 16384) using 64 ranks per node. These tests allowed models for both algorithmic approaches to be produced, considering various parameters of interest to the application team, including FLOP scaling with order, data motion scaling with order, and code scaling with processor count.

3.4.3 CloverLeaf

CloverLeaf [48] is a 2D structured hydrodynamics mini-application that solves the Euler equations using an explicit, second-order method. It was developed to investigate the use of new programming models and architectures in the context of hydrodynamics. We use a modified version of Sod's shock tube [49] as input problem, where we increase the vertical size of the domain and allow the problem to evolve in both dimensions. We ran CloverLeaf in a weak-scaling configuration, where the problem size per node remains fixed. We vary two parameters: the per-node problem size (1, 2, 4, 8, 16) and the number of MPI ranks (64, 256, 1024, 4096, and 16384) using 16 MPI ranks per node. This allowed models to be produced that capture key concerns for CloverLeaf developers: how problem size and processor count impact the scalability of the application.

3.4.4 Evaluation

Real data sets come with separate challenges, such as not knowing the best model, and indeed no guarantees that the assumptions required for our method hold, namely that the optimal model is described by one and only one function and that the function is part of the search space. Fig. 3.3 shows the results of both applying the heuristics and searching the entire solution space. As expected, in the overwhelming majority of cases the two approaches provide the same result (84%), or at least present the same lead-order term (14%). In about 2% of the cases the models differ. The reason is that noise and outliers occurring in real data sets are not limited to any arbitrary threshold. Indeed, it is possible that the effect of noise on performance is larger than the impact caused by the variation of any given parameter. The projection used by the heuristics to generate single-parameter models out of multi-dimensional data diminishes noisy behavior to a higher degree than the exhaustive search does. Therefore, in these rare cases, the heuristic approach results in models with a more conservative growth rate than the ones identified through an exhaustive search. The optimal model is not necessarily the one identified by the exhaustive search, as noise could be modeled alongside the parameter effects.

In all three cases, the model generation for an entire application took only seconds (cf. Fig. 3.4) and was at least a hundred times faster than the exhaustive search. Generating performance models for an entire application means one model per call path and target metric. The search space reduction in all three cases was five orders of magnitude (from 4,250,070 model hypotheses down to 66 per call path and target metric).

3.5 Application insights

In the following, we present the type of insights our approach can deliver. In two case studies, Kripke and BLAST, we look at how performance modeling with multiple parameters can help developers understand and validate the behavior of an application.

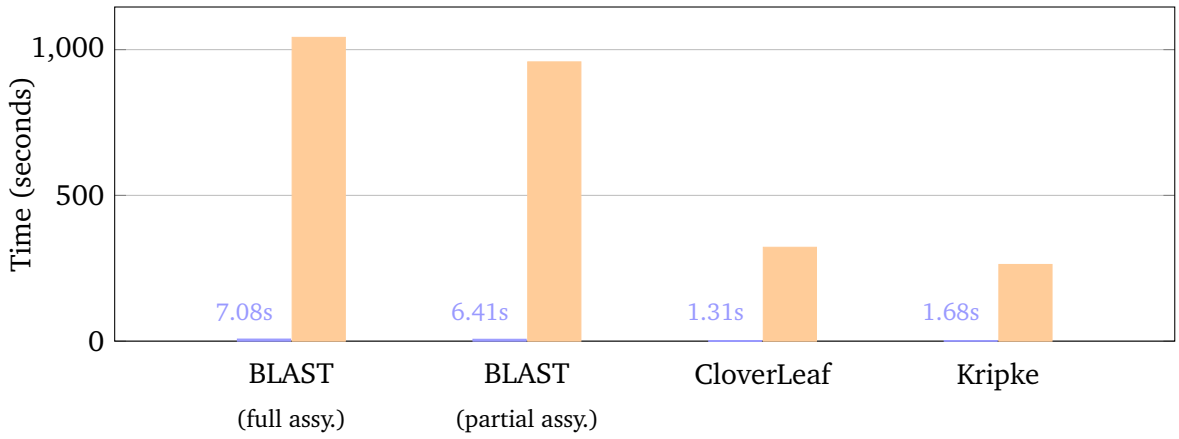


Figure 3.4: Time required to obtain performance models of scientific applications via heuristics (left bar) and via exhaustive traversal of the entire search space (right bar).

3.5.1 Kripke

For Kripke we now increase the number of model parameters to three, enabling more complex behavior to be captured. The number of directions per set and number of groups per set are varied as in Section 3.4. In addition, we vary the number of MPI ranks (8, 64, 512, 4,096, and 32,768). Each rank has 8 OpenMP threads, which means the rank counts correspond to using 1, 8, 64, 512, and 4,096 nodes on Vulcan, using all 64 hardware threads available on a node. These parameter settings represent a realistic range for actual use cases, while remaining tractable. Although we assume that system noise affects our Blue Gene system to a lesser degree, we repeat each test five times to verify its impact experimentally. We ran 750 tests (150 different parameter settings times 5 repetitions each). We determined the confidence intervals and found that there is little to no noise. Had we have known for sure that the system has little to no noise, 150 measurements would have sufficed. This is also true if measurements are restricted to deterministic countable metrics.

The analysis of Kripke covers three parameters: the number of MPI ranks p, the number of directions per direction set d, and the number of groups per group set g. We are particularly interested in the behavior of the `LTimes`, `LPlusTimes`, and `SweepSolver` kernels, as the combination of these three kernels encapsulate the physics simulated by Kripke. Table 3.2 lists selected performance models we generated for these kernels. The `LTimes` kernel computes the spherical harmonic moments of the the angular flux for each element in each group and for each direction. Given that in our weak scaling experiments the number of elements per rank is kept constant, we expect the number of floating-point instructions per rank to remain constant as well. However, we should discover linear relationships with respect to both directions and groups, and their effects to be multiplicative. The model we found is $5.4 \cdot 10^6 \cdot d \cdot g$. $\hat{R}^2 = 1$ indicates that this model is exact. Given the expected availability of floating-point processing power on current and future supercomputers, linear growth is not necessarily a bottleneck, but the combined influence of the two parameters could become challenging. The kernels `LPlusTimes` and `SweepSolver` are structured similarly to the `LTimes` kernel, except that

Table 3.2: Selected multi-parameter performance models for different kernels of Kripke and BLAST.

Metric	Model	$\hat{R}^2$
Kripke		
LTimes		
Floating point instr. [10^6]	$5.4 \cdot d \cdot g$	1
LPlusTimes		
Floating point instr. [10^6]	$5.4 \cdot d \cdot g$	1
SweepSolver		
Floating point instr. [10^6]	$2.16 \cdot d \cdot g$	1
LTimes		
Time [seconds]	$12.68 + 3.67 \cdot 10^{-2} \cdot d^{5/4} \cdot g$	0.989
LPlusTimes		
Time [seconds]	$9.82 + 9.62 \cdot 10^{-3} \cdot d \cdot g^{3/2}$	0.991
SweepSolver		
Time [seconds]	$4.91 + 4.83 \cdot 10^{-3} \cdot p^{1/3} \cdot d \cdot g + 0.90 \cdot d \cdot g$	0.994
MPI_Testany		
Time [seconds]	$6.81 + 0.8 \cdot p^{1/3} + 4.76 \cdot 10^{-3} \cdot p^{1/3} \cdot d \cdot g$	0.996
SweepSolver		
Bytes sent =recv. per msg.[10^6]	$4.8 \cdot d \cdot g$	1
SweepSolver		
Msg. sent = Msg. received	$11250 + 900 \cdot log(p)$	1
BLAST – full assembly		
MPI_Isend		
Bytes sent=recv. per msg.	$1.95 \cdot 10^4 + 81.8 \cdot \log p \cdot o^{7/4} + 4.62 \cdot 10^3 \cdot o^{7/4}$	0.999
BLAST – partial assembly		
MPI_Isend		
Bytes sent=recv. per msg.	$7.63 \cdot 10^3 + 1.31 \cdot 10^2 \cdot \log p$	0.871

the calculations in their innermost loop are different. Nonetheless, as far as the number of floating-point instructions is concerned, all three kernels belong to the same complexity class.

However, the `SweepSolver` kernel's runtime model is different from the other two kernels: The number of parallel MPI ranks appears in the model. The difference stems from the fact that in addition to floating-point calculations, the `SweepSolver` uses MPI to pass data between ranks and ensures that dependencies between ranks are maintained. Theoretically, the

processor count should not affect the number or size of MPI messages sent by each processor, except for a logarithmic term in the message number due to optimizations in the inter-processor communication scheme. The models we have generated are in agreement with this theory and indicate that the $p^{1/3}$ term is caused by waiting on other processors, as shown by the model of the `MPI_Testany` function. The `MPI_Testany` function is called from `SweepSolver` using spin waiting. The $p^{1/3}$ term stems from the three dimensional data decomposition across processes. It represents the diagonal of the process cube, and the waiting time caused by the wavefront traveling along it. The spin waiting causes the performance of these two kernels to compound each other. This is why both kernels show a much smaller $p^{1/3} \cdot d \cdot g$ term, (about 2 orders of magnitude smaller than the lead-order term), representing the interaction caused by the spin-waiting.

Bailey and Falgout [50] show that the theoretical lower bound on the `SweepSolver` kernel for 3D simulations is $\mathcal{O}(p^{1/3}+d\cdot g)$. The key difference between the theoretical lower bound and Kripke's actual runtime performance is the small multiplicative effect caused by the spin waiting. Although the coefficient is quite small, the contribution could become more pronounced at larger configurations.

Since the study above considers three parameters, relying on an exhaustive search would not have been a competitive option. The model generator would have taken more than five hundred years. In contrast, our heuristics-based model generation took less then a minute. This corresponds to a search space reduction of twelve orders of magnitude.

3.5.2 BLAST

BLAST [46] has provided us with the opportunity to study the effects of a parameter, the order, that does not define the input problem size and analyze its interaction with the processor count. When used with a fixed number of degrees of freedom, order is independent of problem specification. That is, for a fixed processor count, changing order does not meaningfully change initial conditions of the simulation, nor the resolution of the degrees of freedom within the mesh. Changing order does change the calculations used within the simulation and the flexibility of the mesh (how likely the mesh is to tangle). In general, a higher order increases the number of calculations and increases the flexibility of the mesh. A discussion of balancing the costs and benefits of increasing order are beyond the scope of this paper. We used the same setup and measurements as in Section 3.4.

When modeling the two different algorithms for BLAST, we have gained new insights into how their parallel communication behavior differs. We model the bytes sent and received in non-blocking fashion and display the results in Table 3.2. We observe that both approaches grow logarithmically with the number of processors in weak scaling mode. The order of the solver has no effect on the partial assembly algorithm but a significant effect on the full assembly algorithm, as indicated by the $o^{7/4}$ component. The developer analyzed our result, and did not expect the order to have such a pivotal effect, or that the order should have such a different effect on the two algorithms. These insights will help the developers better run and optimize the code as they are now aware of the additional cost the order has in full assembly mode. This

result also showcases the compounding effect on performance that parameters have and the need to understand their interactions.

3.6 Discussion

The evaluation with synthetic and real data demonstrates that our heuristics can offer results substantially faster than an exhaustive search—without significant drawbacks in terms of result quality. For three or more parameters, the size of the search space would have prevented such a comparison altogether, which also means that the exhaustive search presents no viable alternative beyond two parameters.

4 Requirements engineering using performance modeling

The last main contribution of this work is a method showing how automated performance modeling can be used to quickly predict application requirements for varying scales and problem sizes. Following this approach, determining the exascale requirements of five major codes becomes possible. We can derive an optimization strategy, and illustrate system design trade-offs in the light of their (optimized) requirements.

4.1 Motivation

Ever-growing computational demands from domains such as climate science, theoretical physics, and neuroscience require large-scale machines in the near future. Planning the design and detailed configuration of such systems is a daunting task since they are often major investments, up to half a billion dollar over their lifetime. Thus, it is extremely important that the machine efficiently supports the execution of all target applications. Designing it is a multi-year planning effort while it stays "top of the line" only for three to five years after installation. Thus, while the machine must be tailored to its workload, it must also be productive from day one on.

Co-designing applications and the system is a powerful technique to ensure early and sustained productivity as well as good system design. In their early phases, such co-designs often rest on *back-of-the-envelope* (BOE) calculations. For example, BOE calculations have been famously used to determine the well-known "bytes-to-flop ratio" for the network and memory in early Cray machines. They continue to gain popularity with requirements-balance models such as the roofline model [51]. In general, such calculations allow problems in applications to be detected early on and their severity to be determined years before the machine is installed or the first prototype becomes available. This is increasingly important since mitigating such problems can often take several person-years. On the system side, BOE calculations allow designers to adjust system parameters to target applications, for example, they can be used to determine the required bytes-to-flop ratio of memory, network, or even the file system. In addition, they can be used to determine required memory sizes, usability of accelerators and co-processors, and even the number of sockets and size of shared-memory domains in the target system.

We automate these BOE calculations in a lightweight requirements analysis for scalable parallel applications. Combining standard performance profiling [13, 52] and stack-distance sampling [53] with an extremely lightweight automatic performance-modeling method [24, 25], we generate empirical models that allow projections for different numbers of processes and problem sizes. Application requirements can be anything such as the number of floating point

operations, the number of bytes transmitted across the network, the number of memory accesses including access locality, or memory consumption. System designers can use the combined process-scaling and problem-scaling models and the specification of a candidate system to determine the resource usage of an application execution with a certain problem size.

Once empirical models are established for an interesting set of requirements, the designer can use them to "play" with configurations such as (1) the amount of memory per node, (2) the speed of memory per node, (3) the network injection speed (how many adapters per node), or (4) the number of cores per socket or node (e.g., many- vs. multicore) etc.. Given the degree of automation we provide, the number of applications and system design choices to be included in the co-design process can be much higher than in a manual study, substantially expanding its breadth.

In this chapter, we demonstrate the power of our technique by analyzing a set of relevant applications and the appropriateness of various system designs. The major contributions of this chapter are:

- The integration and extension of existing performance tools [13, 52, 53, 24, 25] that allows the requirements of parallel applications to be modeled, including memory consumption and access locality
- A technique for practical co-design that extrapolates application requirements to an envisioned system and points out possible bottlenecks on both sides
- A case study with five applications that shows how they would respond to relative system upgrades and how well they would match three different exascale candidate systems

4.2 Requirements engineering

As the foundation of our approach, we define a very simple notion of requirements that supports their quantification in terms of the amount of data to be stored, processed, or transferred by an application. Knowing these numbers alone does not yet allow a precise prediction of application performance or system utilization but can serve as an indicator of the relative importance of certain system resources and how this ratio changes as we scale a program to a larger system. Ultimately, our requirements are expressed in the form of empirical models that allow projections for different numbers of processes and problem sizes.

Our notion of requirements is purely application centric, that is, it does not make any assumption about the hardware other than the ability to run the code as is. Hence, all our requirement metrics refer to data flow at the interface between hard- and software – not between lower layers of the hardware. While the expressiveness of our requirement models is certainly limited, a significant advantage is the simplicity and low effort with which it can be instantiated for a given execution configuration. All that is needed is an MPI implementation of the code, established profiling tools, such as Score-P [13], PAPI [52], and Threadspotter [53], and the performance modeling tool Extra-P [24, 25] to turn the collected requirement data into models. Threadspotter was modified to extract the data it collects for post-processing. Disregarding

Table 4.1: Requirement metrics.

Resource	Metric
Memory footprint	# Bytes used (resident memory size)
Computation	# Floating-point operations (#FLOP)
Network communication	# Bytes sent / received
Memory access	# Loads / stores; stack distance

slight variations in the platform-dependent semantics of certain hardware counters, the requirements of an application can basically be obtained on any system. Since we regard thread-level concurrency not as a requirement in its own right but rather as a way to satisfy requirements, not even specific threading hardware such as a GPU is essential. Likewise, we consider execution time and energy consumption as manifestation of requirement fulfillment and not as their expression. Therefore, the metrics we acquire can be narrowed down to—in most cases—highly reproducible hard- and soft-counters such as floating-point operations or bytes injected into the network.

4.2.1 Application requirements

Since it is currently the predominant programming model and also expected to be highly influential in the future, we stipulate that of each target application an MPI version exists. Application requirements are expressed as a set of functions $r(p, n)$ that predict the demand for resource r depending on the number of processes p and the problem size per process n.

Currently, we consider the requirement metrics listed in Table 4.1, classified by the resource they refer to. Our metrics characterize application requirements in terms of space (i.e., memory consumption) and "data metabolism" (i.e., bytes processed in floating-point units or exchanged via memory and network). Because the amount of data moved between processor and memory subsystem alone is barely a reliable indicator of the pressure an application exerts on the memory subsystem, we also consider memory access locality. Specifically, we capture the stack distance [53] of memory accesses, which is the number of accesses to unique locations that occur between two accesses to the same location.

All metrics refer to a single process, since the matching hardware resources such as CPUs, memory, and network links grow roughly proportionally with the the number of processes expected on a machine. We acquire the metrics related to computation and communication using the Score-P profiler, which captures them at the granularity of individual function call paths, henceforth called *kernels*. This allows bottlenecks to be precisely attributed to individual program locations. Given that the number of floating-point operations required per process is roughly independent of the number of threads used to compute them, we profile the application single threaded, which simplifies our workflow and makes it more robust. We further invoke getrusage() to determine the resident memory occupied by each process across its entire lifetime.

The number of memory accesses and their stack distance is measured using a combination of Threadspotter and PAPI. Originally designed as an interactive locality optimizer for non-expert

users, Threadspotter collects memory access distances only internally to derive optimization suggestions. However, we have modified it such that we can access these metrics directly. Threadspotter identifies loops in an application and instruments groups of instructions that access the same memory location within those loops. Therefore *instruction groups* represent the granularity at which distance metrics are provided. To keep the runtime dilation within practical limits (roughly a factor of eight), Threadspotter samples the execution in short bursts where all memory accesses are documented, followed by periods during which no measurements are gathered. The stack distance used in our approach is a refinement of the more widely known reuse distance [54]. While the reuse distance counts all memory accesses between two consecutive memory accesses to the same location, the stack distance counts only accesses to unique locations, making it a more realistic measure of locality. Since Threadspotter does not count memory accesses, we let PAPI measure the number of load and store instructions for the entire program. Then, we estimate the number of memory accesses per instruction group based on the ratio of samples collected for different instruction groups.

To offset the non-determinism and possible variance of this process, we propose the following methodology to analyze memory locality information: First, any instruction group with less than 100 samples gathered for each measurement configuration is ignored, as the risk of outliers adversely affecting the resulting model is too high. We have observed that both the number and magnitude of outliers is much greater when examining memory locality than it is for other metrics. This becomes obvious when considering a loop which is executed multiple times during the runtime of a program. In the loop itself, stack distance is low if it shows good locality. However, many memory accesses can happen between different executions of the loop, leading to higher stack distance when returning to the loop later on. To capture the most common behavior, we model the median over all gathered samples.

Using the above methodology, we can determine whether memory locality changes and especially whether it drops as the application is scaled up. Careful algorithm designers will use locality-preserving techniques such as tiling or blocking to keep the locality independent of the problem size. Everything else is usually considered a performance bottleneck. With our approximate models, we are able to tell if such a bottleneck exists. If not, we assume that the number of main memory accesses scales with the number of retired load and store instructions, which we can easily measure. We chose distance metrics instead of cache misses because they are hardware independent, less prone to noise and jumps, and therefore easier to model.

4.2.2 Scaling strategy

Our goal is to let an application calculate the largest possible input problem as efficiently as possible. This objective corresponds to the idea of *heroic* runs [55]. Because the overall memory requirement of an application usually grows with the problem size, we therefore expect that an application will occupy all nodes of the machine. Deriving our scaling strategy now becomes straightforward. For simplicity, we assume that the number of MPI processes is determined by the number of processor sockets, so we just need to scale the problem size per process until we exhaust the available memory. Using our memory-consumption model, this is quickly

accomplished. After this point, we can employ our models to predict application requirements on the target system.

4.2.3 Co-design method

Our co-design method is tailored to the early stages of machine procurement when a first vision of the target system needs to be developed and no prototype hardware is available yet. Typical questions to be answered include: Should we rather buy a larger number of thin nodes or a smaller number of fat nodes? Does fat mean more memory or also more floating-point performance through the provision of accelerators or both? If we buy our system in two tranches, which scaling problems should our application developers expect once we upgrade to the full system? Which scaling problems of our workload should we tackle to reconcile conflicting requirements of different codes?

On a more abstract level, the type of question that we can answer is of the following form. How do the requirements of an application change when the application is ported from an old system to a new system and would the new system relax or even tighten existing bottlenecks of the application? In a broader sense, we can tell how well the new system would satisfy the new requirements under the assumption that the old system exactly matched the old requirements. Matching means here that application requirements and system resources are balanced such that the application exhausts all system resources equally. In brief, we always compare two systems in the light of the requirements changes caused by scaling the application from the first to the second.

4.2.4 Model generation

Originally developed to uncover scalability bugs in applications with a large code base [24, 25], the model generator Extra-P requires a set of performance profiles as input, representing runs with different numbers of processes and problem sizes. The input sources used in this study encompass performance profiles obtained with PAPI, Score-P, and Threadspotter.

As a rule of thumb, we need to run measurements for at least five different configurations of each parameter we consider, requiring 25 measurements in the case of process count and problem size variation. The output of the generator is a set of human-readable functions, one for each instrumented program location and metric. Each function describes the evolution of the metrics as the number of processes and the problem size per process are changed.

Different from previous studies [24, 25], however, we refrain from modeling execution time. Time is suitable to find certain classes of scalability problems, but is affected by noise and all sorts of other non-linear effects. Most important, however, it is highly architecture dependent, which means that the results are generally not portable to a new system and, thus, of little value for its design. Our primary goal is the characterization of application behavior, which is why we restrict ourselves to hardware and software counters that represent the application and not a specific hardware (cf. Section 4.2.1).

4.3 Application requirements study

We demonstrate our requirement modeling method with five applications: Kripke [45], LULESH [56], MILC [22], Relearn [57], and icoFoam [58]. The first four can be considered realistic candidates for exascale deployment. We gathered our measurements on two test systems: the first one is an IBM Blue Gene/Q with almost 500,000 cores. Each node features one PowerPC A2 processor with 16 cores running at 1.6 GHz. The second is a Linux cluster that consists of 706 nodes with two 8-core Intel Xeon E5-2670 processors on each node, running at 2.6 GHz. We obtained the measurements for LULESH, MILC, and Relearn on the BG/Q system and those for icoFoam and Kripke on the Linux cluster. Because Threadspotter does not support the processor of the BG/Q system, we conducted these measurements on the Linux cluster for all applications.

The following subsections describe the most important findings for each application. We also discuss the most likely bottlenecks, leading to suggestions as to which requirements need to be optimized. We generated models for the previously described metrics by gathering measurements for combinations of process count and problem size per process. Our results are summarized in Table 4.2 and discussed below.

4.3.1 Kripke

Kripke is a 3D Sn particle transport code. It is written in C++ and implements an asynchronous MPI-based parallel sweep algorithm. A major goal of Kripke is the evaluation of programming models, data layouts, and sweep algorithms in terms of their performance impact. In our test runs, we varied the problem size, defined as the simulated volume per process, and the number of processes.

Requirements In Table 4.2 we see that computation and communication are linearly affected by the problem size per process, while the process count shows no effect at all. Memory locality is unaffected by both problem size and process count. The number of load and store instruction however grows with the product of problem size and process count. Similar to computation and communication, we only discover a linear growth of the memory footprint when increasing the problem size per process.

Conclusion The requirement that would benefit most from optimization is memory access. Transforming the multiplicative impact of process count and problem size on the number of loads and stores into an additive one through algorithmic improvements would remove the last performance bottleneck, as seen in Table 4.2. We conclude this analysis by stating that Kripke should scale to a large number of processes, but the growing number of memory accesses will eventually lead to a slowdown.

4.3.2 LULESH

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) code was one of five challenge problems in the DARPA UHPC program and has since become a widely studied proxy application in DOE co-design efforts for exascale. This makes it an excellent candidate for our evaluation. LULESH is a mini-app which calculates simplified 3D Lagrangian hydrodynamics on an unstructured mesh. It is written in C++ and supports a variety of parallelization paradigms, among which we focus on MPI. In our test runs, the problem size defines the volume of the cube per process.

Requirements. We observe in Table 4.2 that the required number of floating-point operations grows with both problem size and process count, combined in a product expression. While the contribution of the process count alone is only $\mathcal{O}(p^{0.25} \cdot \log p)$, it may still lead to a small scalability issue when the problem size per process grows larger, as indicated by the super-linear relationship $\mathcal{O}(n \cdot \log n)$. The same behavior we notice for computation can be observed for communication as well, although here the problem size has a strictly linear effect. The number of loads and stores is proportional to $\mathcal{O}(n \cdot \log n \cdot \log p)$. However, neither problem size nor process count affect memory locality. The memory footprint is proportional to $\mathcal{O}(n \cdot \log n)$. While not ideal, it will still allow a lot of flexibility when tailoring the problem size per process to fit the available memory on different systems.

Conclusion The growth of all requirements with respect to both problem size and process count is very close to ideal. While a small amount of optimization is possible in this direction, such as eliminating logarithmic growth effects, the biggest potential for improvement lies in transforming multiplicative contributions of problem size and process count into additive ones. This would effectively eliminate even the small performance issues previously discussed. With the current implementation, the multiplicative contribution process count and problem size per process have on computation and communication for LULESH is a small obstacle in tailoring and scaling the application to run on different systems. Their growth is slow enough to limit these issues at anything except the most extreme scales.

4.3.3 MILC

The MILC – MIMD Lattice Computation – tool set is a set of codes for studying quantum chromodynamics (QCD) via parallel simulations of the SU(3) lattice gauge theory on a four-dimensional lattice. MILC is a highly scalable application, which consumes a major fraction of the CPU cycles in US DOE and NSF computing centers. Its scalability makes it a good candidate for early testing of next-generation systems and also a good co-design candidate for an exascale system. We analyze the application MILC/su3_rmd. Its runtime was modeled in previous studies [22, 24]. However, no requirement models exist for MILC. For MILC, we varied the number of processes and the problem size per process expressed through the size of the simulated lattice.

Table 4.2: Per-process requirement models. p denotes the number of processes and n the problem size per process. For each metric, we show the terms with the largest impact on performance for both problem size per process and number of processes across all computed models of each application. The coefficient of each term is the sum across the entire program, rounded to the nearest power of ten. If the constant contribution is relevant for the parameter ranges measured, we also include it in the table. We mark potential performance bottlenecks with a warning sign.

	Requirement	Model	
Kripke	#FLOP	$10^6 \cdot n$	
	#Bytes sent & received	$10^4 \cdot n$	
	#Loads & stores	$10^8 \cdot n + 10^5 \cdot n \cdot p$	⚠
	Memory locality	None	
	#Bytes consumed	$10^5 \cdot n$	
LULESH	#FLOP	$10^5 \cdot n \log n \cdot p^{0.25} \log p$	⚠
		$10^3 \cdot n^{1.5} \cdot \log p$	
	#Bytes sent & received	$10^3 \cdot n \cdot p^{0.25} \log p$	⚠
	#Loads & stores	$10^5 \cdot n \log n \cdot \log p$	
	Memory locality	None	
	#Bytes consumed	$10^5 \cdot n \log n$	
MILC	#FLOP	$10^{10} \cdot n + 10^7 \cdot n \log p$	
	#Bytes sent & received	$10^4 \cdot Allreduce(p)$	
		$10^4 \cdot Bcast(p)$	
		$10^9 \cdot n$	
	#Loads & stores	$10^{11} + 10^8 \cdot n \log n + 10^5 \cdot p^{1.5}$	
	Memory locality	$10^5 \cdot n$	
	#Bytes consumed	$10^6 \cdot n$	
Relearn	#FLOP	$10^3 \cdot n \log n \cdot \log p + p$	
	#Bytes sent & received	$10^5 \cdot Allreduce(p)$	
		$10 \cdot Alltoall(p)$	
		$10 \cdot n$	
	#Loads & stores	$10^6 \cdot n \log n + 10^5 \cdot p \log p$	
	Memory locality	None	
	#Bytes consumed	$10^6 \cdot \sqrt{n}$	
icoFoam	#FLOP	$10^8 \cdot n^{1.5} \cdot p^{0.5}$	⚠
	#Bytes sent & received	$n^{0.5} \cdot Allreduce(p)$	⚠
		$p^{0.5} \log p$	⚠
		$n \cdot p^{0.375}$	⚠
	#Loads & stores	$10^8 \cdot n \log n \cdot p^{0.5} \log p$	⚠
	Memory locality	None	
	#Bytes consumed	$10^3 \cdot n + 10^2 \cdot p \log p$	⚠

Requirements As can be seen in Table 4.2, a few kernels exhibit negligible logarithmic growth of floating-point operations per process, as the process count is increased. This can be for

all purposes considered insignificant, even at exascale and beyond, since the floating-point requirement is overwhelmingly determined by the problem size alone. MILC is implemented in a highly scalable fashion and only MPI collectives, such as allreduce and broadcast with constant message sizes, imply communication demands that grow with the number of processes. Their exact complexities depend on the specific implementation of the collective algorithms [59], and are therefore represented in Table 4.2 by $Allreduce(p)$ and $Bcast(p)$, respectively. The payload of MPI collectives does not depend on the problem size, and has an upper bound in the order of 10^4 bytes. However, we observe a payload growth in non-blocking communication call paths with increasing problem size per process, with a maximum of $10^9 \cdot n$. Increasing the problem size per process linearly increases the stack distance for 60% of all memory accesses. The number of load and store instructions grows independently with both process count and problem size per process. The process count contribution stems from the MPI communications and is several orders of magnitude smaller than the effect of the problem size per process. The model for resident memory size shows a linear contribution of the problem size per process, which is expected for most applications and poses no scalability problem.

Conclusion The requirement than can benefit most from optimization is memory access. If the memory locality was improved, increasing the problem size per process would be possible without losing performance. We conclude this analysis by stating that MILC/su3_rmd should have no issues scaling to exascale and beyond. The problem size per process can only be moderately increased without degrading memory access times. The effects of process count and problem size on performance are independent with the exception of floating-point operations that, however, is insignificant for the overall performance. MILC/su3_rmd should therefore be able to fit most target systems without significant performance loss.

4.3.4 Relearn

Relearn simulates the dynamics of the connectome in the brain, that is, how connections between individual neurons are formed and deleted. This is also called structural plasticity. The code is written in C++ and parallelized with MPI. In comparison to the original version [57], optimized memory management makes the code used in this study far more scalable. The problem size parameter in our tests defines the number of neurons per process to be simulated.

Requirements As can be seen in Table 4.2, the floating-point operations per process grow with $\mathcal{O}(n \cdot \log n)$ as the problem size per process is increased, scaled by the logarithm of the process count. This behavior is the one dominating the computation of Relearn. Moreover, the process count adds another linear contribution, insignificant when compared to the effect of problem size per process. We can only observe how MPI collectives, such as alltoall and allreduce with constant message sizes, show growing communication demands with increasing process counts, which is expected. The payload for MPI point-to-point messages does not depend on the process count, but grows linearly with the problem size. The number of loads and stores grows with both process count and problem size per process. However, neither problem size nor process count affect memory locality. The memory footprint grows only with the square root of the

problem size, allowing Relearn to take advantage of architectures where the available memory forms the bottleneck. While increasing the number of neurons should have a linear effect on the memory required, the data structures used to contain them have a much more significant impact in the ranges measured and projected, but grow only with the square root of the problem size.

Conclusion Relearn has no issue in scaling to any number of processes. The effects of process count and problem size on performance are mostly independent and the exceptions insignificant for overall performance. Furthermore, Relearn will be able to vary the domain size per process to fit multiple target systems without significant performance loss and can accommodate systems where memory is at a premium.

4.3.5 IcoFoam

IcoFoam is a solver in the widely used open-source computational-fluid-dynamics code OpenFOAM [58]. OpenFOAM, developed by ESI/OpenCFD, is a non-monolithic library encompassing over 80 flow solvers that supports numerical simulations of a broad variety of continuum models describing transport processes ranging from fluid flow and chemical reactions to turbulence, acoustics, and electromagnetics. We analyzed the unmodified icoFoam executable from the demonstration instance of OpenFOAM (development version from April 2017 from openfoam.org). This flow solver implements a method suitable for the incompressible flow of a Newtonian fluid under isothermal conditions. We applied the solver to a two-dimensional test case, namely the well-known lid-driven cavity case [60]. The problem size is defined as the number of computational cells per process.

Requirements. Table 4.2 shows that the influence of process count and problem size per process on computation is multiplied. This affects the flexibility with which icoFoam can be mapped onto different architectures. For example, it is not possible to make the problem size per process smaller and increase the number of processes proportionally without changing the computational requirements of icoFoam. Beyond the known dependence of allreduce on the process count, its payload grows with the square root of the problem size per process. The payload of non-blocking point-to-point communications also grows with the product of problem size per process and process count. Both issues represent significant scalability bottlenecks: even if the individual contributions of problem size per process and process count are linear or less, their multiplication makes performance losses more likely. Problem size and process count both contribute to an increase of memory accesses and could lead to issues similar to the behavior discovered for computation. The locality remains unaffected by the varied parameters. Problem size and process count cause independent growth of the memory footprint. While a linear growth with the problem size per process can be expected, the additive effect of the number of processes may limit scalability at some point.

Conclusion. Memory consumption and memory access, communication, and computation limit the scalability of icoFoam in this implementation. The multiplicative combination of pro-

Table 4.3: Process count and memory per process available to applications for three different system upgrade scenarios.

System upgrade	Process count	Memory per process
A: Double the racks	$p' = 2 \cdot p$	$m' = m$
B: Double the sockets	$p' = 2 \cdot p$	$m' = 0.5 \cdot m$
C: Double the memory	$p' = p$	$m' = 2 \cdot m$

cess count and problem size for all requirements except memory footprint means that we can not adapt to system differences without incurring significant performance losses. The severity and multitude of performance issues suggest that a different approach is required as a whole. Our analysis confirms that the icoFoam development version from April 2017 is not a suitable candidate for exascale. Others solvers, such as interFoam, from a different release version of OpenFOAM, however, have been successfully used for petascale simulations [61].

4.4 Co-Design Study

The key point of our method is to *guide the programmer to find application bottlenecks relative to an architecture as well as to guide the architect to find system bottlenecks that a given application would experience.* This study exemplifies our co-design method, which we recommend as the first step in an iterative co-design process. Once a prototype of the target system becomes available and it is more precisely known at which rate the system will satisfy certain requirements, their balance or imbalance can be more accurately determined. Possible modifications to either application or system may entail further iterations.

Specifically, we focus on two questions related to co-design and extreme scaling of applications. The first question we answer is "*Given a large system defined such that the application equally exhausts all available resources, which of the possible upgrades would benefit the application most?*" Possible upgrades we consider are (A) doubling the entire system, (B) doubling the number of processor sockets per node and leaving everything else constant, and (C) doubling the memory and leaving everything else constant. These upgrades and how the resources that are available to the applications change on the new systems are summarized in Table 4.3. The combination of possible upgrades can be expanded to encompass further realistic scenarios that are considered by system architects. This question focuses on relative differences between systems. The second question we answer is "*How would the application performance change on different proposed exascale systems?*" This question sheds light on how differences in system design affect the studied applications by looking at absolute numbers rather than relative differences.

Although we concede that exceptions may exist, often encountered difficulties of exploiting more thread-level concurrency than a single socket provides lead us to the assumption that each socket will run a separate and potentially multithreaded MPI process. Nonetheless, systems with monolithic nodes like Blue Gene/Q can still be accurately modeled by adapting the definition of a socket to the most common usage scenario. Since the number of cores only matter as far as

they are translated into parallel speedup, we refrain from specifying an exact number of cores, also because they may not necessarily be homogeneous.

Table 4.4: Workflow for determining the requirements of application App after doubling the number of racks (upgrade A).

I Create requirement models for memory footprint, communication, computation, and memory access of App.

Example

Metric	Process scaling and problem scaling
Computation	$p \cdot n^2$
Communication	$p \cdot \sqrt{p}$
Memory access	$\sqrt{n}$
Memory footprint	$p \cdot n$

II Determine the new maximum number of processes and new memory available per process that the upgraded system supports.

Example

Configuration parameter	Old	New
# Processes	p	$p' = 2p$
Memory	m	$m' = m$

III Determine the new memory footprint requirement per process if all processors are used.

Example

Metric	Old	New
Memory footprint	$n \cdot p$	$n \cdot 2p$

IV Determine the new problem size per process such that the memory footprint equals the memory available to each process and compute the new overall problem size.

Example

Metric	Old	New	Ratio
Problem size	$n = m/p$	$n' = m/2p$	0.5
Overall problem size	$p \cdot n$	$p' \cdot n'$	1

V Determine the new requirements for computation, communication, and memory access.

Example

Metric	Old	New	Ratio
Computation	$p \cdot n^2$	$2p \cdot 0.25n^2$	0.5
Communication	$p\sqrt{p}$	$2p\sqrt{2p}$	$2\sqrt{2}$
Memory accesses	$\sqrt{n}$	$\sqrt{0.5 \cdot n}$	$\sqrt{0.5}$

Table 4.5: System upgrade comparison. We show how each requirement of an application changes in response to each upgrade. High values indicate scalable behavior for the problem size per process, whereas low values indicate scalable behavior for everything else. The desired ratio between the new and old systems, which is the same for all requirements, is provided once for each upgrade.

Ratios \ Applications	Kripke	LULESH	MILC	Relearn	icoFoam
System upgrade A: Double the racks			Desired ratio:		1.0
Problem size per process	1	1	1	1	0.5
Computation	1	1.2	1	1	0.5
Communication	1	1.2	1	1	0.7
Memory access	2	1.2	2.8	2	0.7
System upgrade B: Double the sockets			Desired ratio:		0.5
Problem size per process	0.5	0.5	0.5	0.3	0.3
Computation	0.5	0.6	0.5	0.3	0.2
Communication	0.5	0.6	0.5	0.3	0.3
Memory access	0.5	1	1.4	1	0.5
System upgrade C: Double the memory			Desired ratio:		2.0
Problem size per process	2	1.4	2	2.8	1.4
Computation	2	1.4	2	2.8	1.7
Communication	2	1.4	2	2.8	1.4
Memory access	2	1.4	2	2.8	1.4

4.4.1 System upgrade

To exemplify the process of determining how an application would respond to a system upgrade, we choose system upgrade A, that is, doubling the racks of the system. We enumerate and describe the different steps of our scaling method in Table 4.4. The notional requirements of application App are listed at the top of the table as part of step I. Following this process, we can easily draw conclusions regarding system utilization, requirements balance, and usefulness of a particular upgrade. The ratios between new and old problem sizes indicate how the largest problem size that can be solved changes, both per-process and overall. The ratios between new and old requirements indicate which system components will experience an increased load relative to other components.

Due to the increasing memory footprint, the total problem size cannot be doubled in this case, as one would desire. In fact, the overall problem size has to remain constant, as the memory requirement is the product of problem size per process and number of processes. The computational requirement decreases due to the quadratic effect of the problem size per process.

Furthermore, the network requirement grows to be almost three times than what it was originally, likely leading to network congestion and a significant slow-down. The number of memory accesses, a function of only the problem size per process, is reduced in this configuration.

In conclusion, our hypothetical application App can barely profit from the the system upgrade. It can exploit increased parallelism, halving the number of floating-point operations required per process, but not solve a larger overall problem. Also, the communication demand per process grows by more than a factor of two, potentially leading to network congestion. After having demonstrated the types of insights our method delivers with an exaggerated fictional example, we now look at real codes.

Using the five applications studied, namely Kripke, LULESH, MILC, Relearn, and icoFoam, we now apply the previously illustrated workflow to analyze the benefits and drawbacks of different system upgrades. We use the requirement models determined for each application individually, as well as the upgrades listed in Table 4.3 to determine the new problem sizes per process for each application, assuming that all processors a system provides are used. We then determine the new requirements for computation, communication, and memory access. Our comparative analysis is numerically summarized in Table 4.5. We consider an optimistic linear relation between problem size per process and requirements as a baseline for scalability. For example, if we double the racks we wish that the total problem size that can be solved should double, too, but that the requirements per process remain the same. This simplifying assumption will not be generally true, but provides a notion of desirable behavior for our discussion.

Apart from MILC, no other application has shown any change in memory locality with respect to process count and problem size per process. We therefore focus on the total number of load and store instructions as the primary memory-access metric in these cases.

When analyzing Table 4.5, it becomes obvious that the most important parameter is the problem size per process, as it determines all other requirements and how well the stated goal of trying to perform heroic runs is met. Our goal is to do more science, to run larger simulations, not just to run them faster. While Kripke and MILC provide no surprises, all other applications have non-linear memory requirements with varying problem sizes and process counts. This leads to different total problem sizes that can be tackled when the ratio between the number of processes and the memory available to a process changes, even if their product is the same.

For example, when doubling the racks the problem size per process should remain constant, meaning that the biggest possible problem size that can be solved on the upgraded system should be two times larger than the original one. This is true in our analysis for all applications except the development version of icoFoam. Since the problem size per process is only 0.5, this means the studied instance is limited to effectively solving the same overall problem size on the new system.

Relearn on the other hand has a memory footprint that only grows with the square root of the problem size. This means that when less memory per process is available, its maximum problem size per process will be smaller than desired, as is the case for system upgrade B, where its problem size per process is only about a third of the original, while the desired would be half of the original problem size per process. Conversely, in the case of system upgrade C,

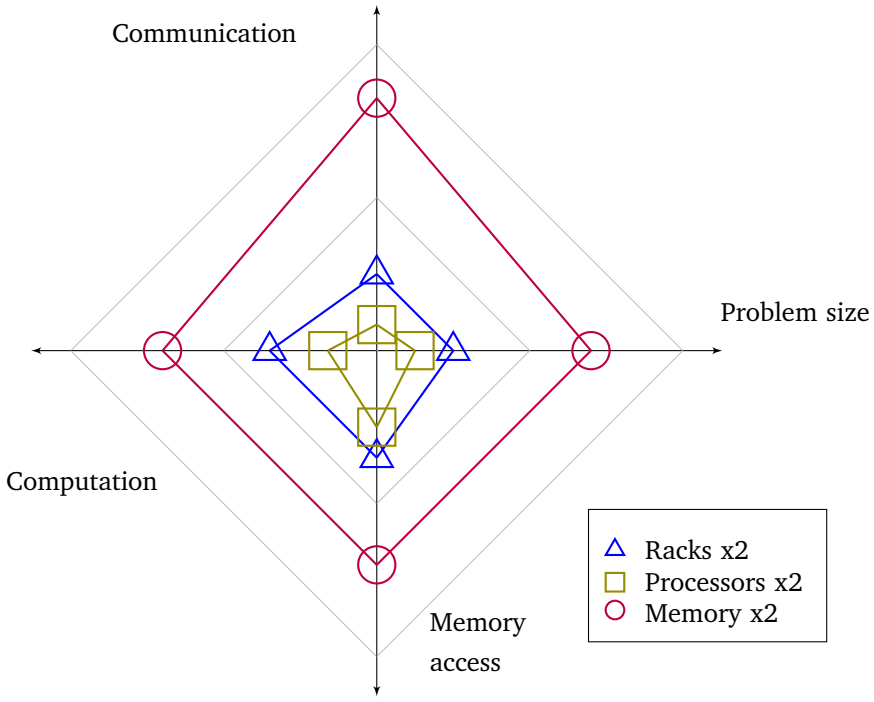


Figure 4.1: Requirements of icoFoam after different system upgrades.

where the memory per process is doubled, its maximum problem size grows to be larger than the desired two times, and becomes almost three times the original value.

When considering the computation, communication, and memory access requirements per process, these should ideally follow the same behavior as the problem size per process. None of the analyzed applications reach this ideal, although Kripke and MILC come close with only one and two deviations, respectively, which one can see by tracing their columns in Table 4.5.

The other aspect is performance, as simply being able to handle a larger problem size does not imply that the machine is used to its full potential. Just increasing the available memory and the problem size per process along with it, will likely cause the overall performance of all applications to drop as the requirements will grow and might exceed available resources, causing bottlenecks to appear. For example, Kripke and MILC will double their per-process requirements for computation, communication, and memory access in system upgrade C, as one would expected. If any of the corresponding hardware resources become over-utilized, performance will degrade.

In Figure 4.1, we visualize the information from Table 4.5 for icoFoam, one of our test cases, in a radar plot. The radar plot allows requirements imbalance within an application to be more easily spotted. Whenever one node of the diamond in the radar plot is located further from the center than others, it signifies a requirement growing faster than the others. The graphical representation obscures the actual ratios, but offers a more intuitive and faster way to compare the requirement models of a given application. We believe both the visual and the numeric representation complement each other, which is why we recommend to use them in tandem.

The first lesson we can learn from this analysis is that getting the best performance is not synonymous with solving the largest problem. The second is that the ratio of problem size per process to process count is of major importance for applications where the growth requirement for memory is different between process and problem scaling.

4.4.2 System design

Now, we investigate how the applications studied would map to potential exascale straw-man systems. Rather than using relative upgrades as in the previous section, where we assumed that certain characteristics of an existing system are doubled, we now focus on how our method works when applied to absolute values for system characteristics such as FLOPS per processor. There are a number of hardware architecture directions suggested to reach exascale. Major differences between the approaches lie in their ratio of nodes to processors to FLOPS per processor, which combined are supposed to reach 1 exaFLOPS. By processor we define a computational unit designed to run a process (potentially multi-threaded). Possible design options for such systems are presented in Table 4.6 and summarized below:

- A **massively parallel system** that consists of a large number of nodes with many but weak processors
- A **vectorized system** that consists of a small number of nodes with few but very powerful processors
- A **hybrid system** with even less nodes, but with a sufficiently large number of moderately powerful processors

For this study, we assume a total memory per system of 10 PB, divided equally among all processors. This value is consistent with the rates of FLOPS to memory of current top supercomputing systems in the world.

The total memory, IO, and network resources can also vary, but more likely as a function of the available funds and not to satisfy a certain ratio to other resources. A more detailed analysis where more system characteristics would vary is certainly possible, but would not be qualitatively different.

For this analysis, only the computational requirement and memory footprint relative to the problem size per process and the number of processes are taken into consideration. We determine the number of processes for each of the systems by multiplying the node and processor counts, as we want to have access to the full exaFLOPS. For each application we can then determine the problem size per process that would consume all the memory available to a process. Knowing the problem size per process and the number or processes, we determine the overall problem size for each application. The results of this workflow, which is similar to the one presented in Table 4.4, are presented in Table 4.7.

IcoFoam, more precisely the solver from the development version of OpenFOAM from April 2017, is notably absent in Table 4.7, as the number of processes adversely affects the memory required per process. This unfortunately means that the studied instance of the code cannot fully utilize any of the three systems, as the memory requirement regardless of problem size per

Table 4.6: Characteristics of three exascale straw-man systems.

Metric	M. parallel	Vector	Hybrid
Nodes	$2 \cdot 10^4$	$5 \cdot 10^4$	10^4
Processors	$2 \cdot 10^9$	$5 \cdot 10^7$	10^8
Processors per node	10^5	10^3	10^4
Memory per processor	$5 \cdot 10^6$	$2 \cdot 10^8$	10^8
FLOPS per processor	$5 \cdot 10^8$	$2 \cdot 10^{10}$	10^{10}

Table 4.7: Overall maximum problem size for selected applications and time each application needs to solve the same problem on different exascale straw-man systems described in Table 4.6, assuming perfect parallelization. All metrics with the exception of overall problem size are expressed per process. Following a workflow similar to Table 4.4, we determine the values using the requirement models from Table 4.2.

	Metric	M. parallel	Vector	Hybrid
Kripke	Problem size	50	$2 \cdot 10^3$	10^3
	Overall problem size	10^{10}	10^{10}	10^{10}
	Duration	0.1	0.1	0.1
LULESH	Problem size	17.4	342	190
	Overall problem size	$3.9 \cdot 10^{10}$	$1.7 \cdot 10^{10}$	$1.9 \cdot 10^{10}$
	Duration	40	21.5	33
MILC	Problem size	5	$2 \cdot 10^2$	10^2
	Overall problem size	10^{10}	10^{10}	10^{10}
	Duration	10^2	10^2	10^2
Relearn	Problem size	25	$4 \cdot 10^4$	10^4
	Overall problem size	$5 \cdot 10^{10}$	$4 \cdot 10^{12}$	10^{12}
	Duration	4	0.02	0.2

process is larger than what is available if all processors are used. While it would be possible to run this code on a smaller subset of processors, that is not the focus of our study. InterFoam, another solver from a different version of OpenFOAM, has recently shown a better potential for scalability and will be the focus of future investigations [61].

We discover that for Kripke and MILC the different system types do not affect the largest overall problem size that can be solved. That is because any difference in the ratio between process count and problem size per process can be offset by configuring the application appropriately. The situation is different for the other applications, as the ratio p/n between process count and problem size is more relevant to the required memory. Relearn can solve much larger overall problems on a system with fewer but stronger processors, while LULESH can solve the largest problem on the massively parallel system.

Being able to solve a larger problem is very useful, but we also wish to see on which system a given problem can be solved faster. Towards this goal, we take the biggest overall problem size

of each application that can be solved on all systems, and change the problem size per process such that each system solves the same problem. We still use all available processors to have access to all computational resources each system provides. We can then use the problem size per process and the number of processes to determine the number of floating-point operations required per process. After dividing this requirement by the floating-point rate offered by the processor, we can estimate a lower bound of the runtime this computation takes. The lower bound is based on the simplifying assumption that parallelization is perfect and no communication overhead exists. The smaller the duration we obtain, the more efficiently the application may use the system in question, possibly offering better performance. The results of these steps of the workflow are presented in Table 4.7. Similar to the overall problem size, the times Kripke and MILC take to solve the problem are the same on each system. However, both Relearn and LULESH benefit more from a high ratio and would perform better on the vectorized system. To shift the lower bound closer to more realistic runtimes, we need to take other requirements such as communication into account, which is feasible as long as the system designer can specify the rates at which the hardware can satisfy them.

Much better behavior for LULESH could be achieved by optimizing the algorithms such that the effects of problem size per process and process count are additive rather than multiplicative with respect to the different requirements: $\#FLOP = 10^5 \cdot n \cdot \log n + p^{0.25} \cdot \log p$. In the example from Table 4.7, this would improve the overall time to solution by approximately three orders of magnitude on each system, to less than 0.1 seconds compared to between 20 and 40. It would also change how LULESH performs on the different systems, obtaining the best results on the massively parallel system as opposed to the vectorized system, which the unoptimized version favors.

The recommended course of action beyond improving the applications is to experiment with a small number of prototype nodes of the kind to be employed in the system, and determine the actual rates at which the requirements can be satisfied. Requirements other than computation such as memory access and network communication must be considered. For example, similarly to our analysis of potential exascale system candidates with different nodes to processors to FLOPS ratios, an analysis of the network requirements taking network bandwidth, latency, and topology into account can be performed. However, when considering a small number of nodes as opposed to an entire rack or multiple racks, network communication may differ qualitatively. Having a test system that contains at least all types of network connections the full system is supposed to have could be a compromise.

The co-design study in this section can provide additional information about which types of systems provide the best opportunities for a given set of applications. With this data a more informed decision can be made regarding the scalability potential of the application.

4.5 Conclusion

In this chapter, we introduce a quick and simple automatic back-of-the-envelope technique to generate requirement models for parallel applications. The workflow we propose leverages these models to enable system designers to ponder various upgrade and design options. An

important contribution is the ability to compare how the requirements for different resources change in relation to each other when a code is scaled. We demonstrate our lightweight co-design approach using five different applications. Our analysis shows how requirement models can be used to understand potential bottlenecks and how to balance a system configuration to support a certain application, both when considering relative upgrades of a system and when looking at the absolute values of different system parameters. We believe that our method can be easily applied to a full compute-center workload consisting of several dozens of applications.

5 Impact

The contributions presented in this work had a significant impact on the field of automatic performance modeling. They have been used by other researchers in collaboration with the author of this thesis as starting points towards new approaches toward analyzing and improving the performance of HPC applications and systems. Extra-P was used by application developers such as UG4 or Mafia to understand and improve their code performance. Lastly, collaboration with other authors have improved and expanded the capabilities of Extra-P and will be integrated into upcoming releases.

5.1 Scalability framework

Using Extra-P as a foundation, Shudler et al.[3] have developed a scalability framework that helps developers compare their expectations for performance with empirical performance models to uncover limitations in libraries, applications or even underlying platforms. The authors provide a detailed study of MPI implementations on three different hardware platforms, exemplifying the types of insights that can be gained.

5.1.1 Framework overview

The scalability framework, shown in Figure 5.1, builds on automatic empirical model generation. The improvement comes in the form of performance expectations. The main source of performance expectations are the developers themselves, but depending on the use case, previous implementations or other versions of the same code could be used as well. The performance expectations provide an alternative way of search space generation for Extra-P. Instead of relying on a default set of hypotheses, the search space can be generated specifically around the expectation, to provide more detailed options in the immediate vicinity. This allows the method to be more versatile in its results, without increasing the time to solution.

The difference between the empirically generated performance models and the performance expectations, called divergence models forms the core of this approach. It allows a very simple analysis of the performance of the target code. If the empirical model and the expectation are the same, no issues are uncovered. If they are not the same, the authors suggest two different classifications. They decide whether the difference between empirical model and expectation are acceptable or not by using a user-defined deviation limit. The developer would express this limit either in absolute terms of relative to the growth rate of the models themselves. For, example, if the expected model is p^2 and the empirical model is $p^2 \cdot \log p$ the divergence model is $\log p$. If the deviation limit is $\sqrt{p}$, the divergence model above would be acceptable. The advantage of using such an approach is that it permits the classification of issues, and allows developers to focus on the biggest problems first.

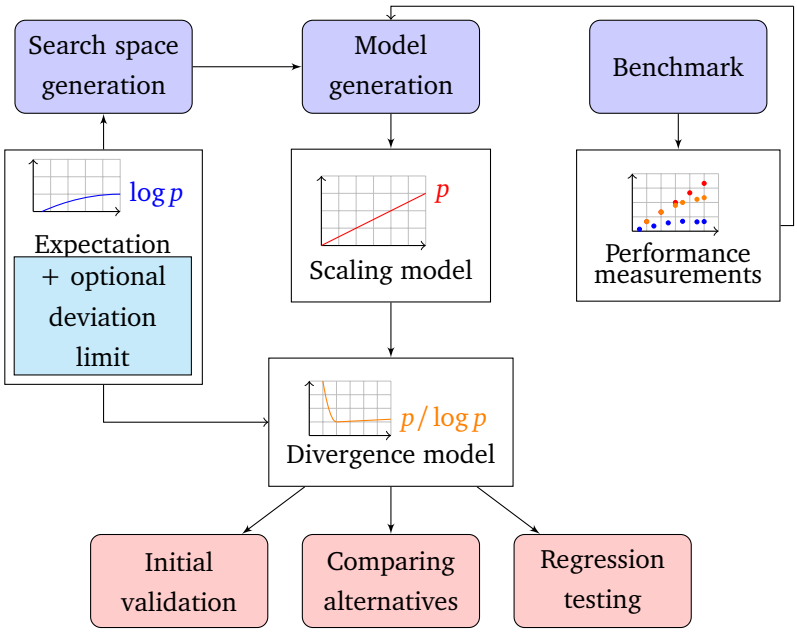


Figure 5.1: Framework overview including use cases [3].

The authors suggest three different use cases for the framework, applicable even in the early design and implementation phases of a project. If the developers write their code with performance expectations in mind, they can use this approach to validate these at every step along the way. Furthermore, if a particular task can be accomplished in multiple ways, each implementation could be modeled separately to choose the best one. It is even possible that different implementations are superior for certain parameter configurations. This would in turn suggest switching the algorithm used depending on parameter configuration to make sure the best performance is achieved. Lastly, by adding the modeling approach to a build-testing system, not only the correctness but also the performance of the code could be constantly examined. Previous performance results could form the expectation for subsequent versions, ensuring that adding features or fixing issues does not degrade performance.

5.1.2 MPI case study

The authors used extensive literature available to compile a list of ideal latency-oriented performance expectations for MPI collectives as well as memory requirements of MPI communicators. They then gathered measurements on the HPC systems Juqueen (Blue Gene Q), Juropa (Intel cluster using InfinyBand interconnects) and Piz Daint (Cray-XC30) and compare the empirical models resulting from these measurements with the theoretical expectations.

The analysis has uncovered significant differences in the performance of the different MPI collective implementations, a selection of which is presented in Figure 5.2, as well as a previously unknown performance issue in the memory requirement of communicator duplication on the Cray system, which the developers were then able to confirm. For example, the models for Barrier are different on each of the three HPC systems analyzed. Overall, the implementation on

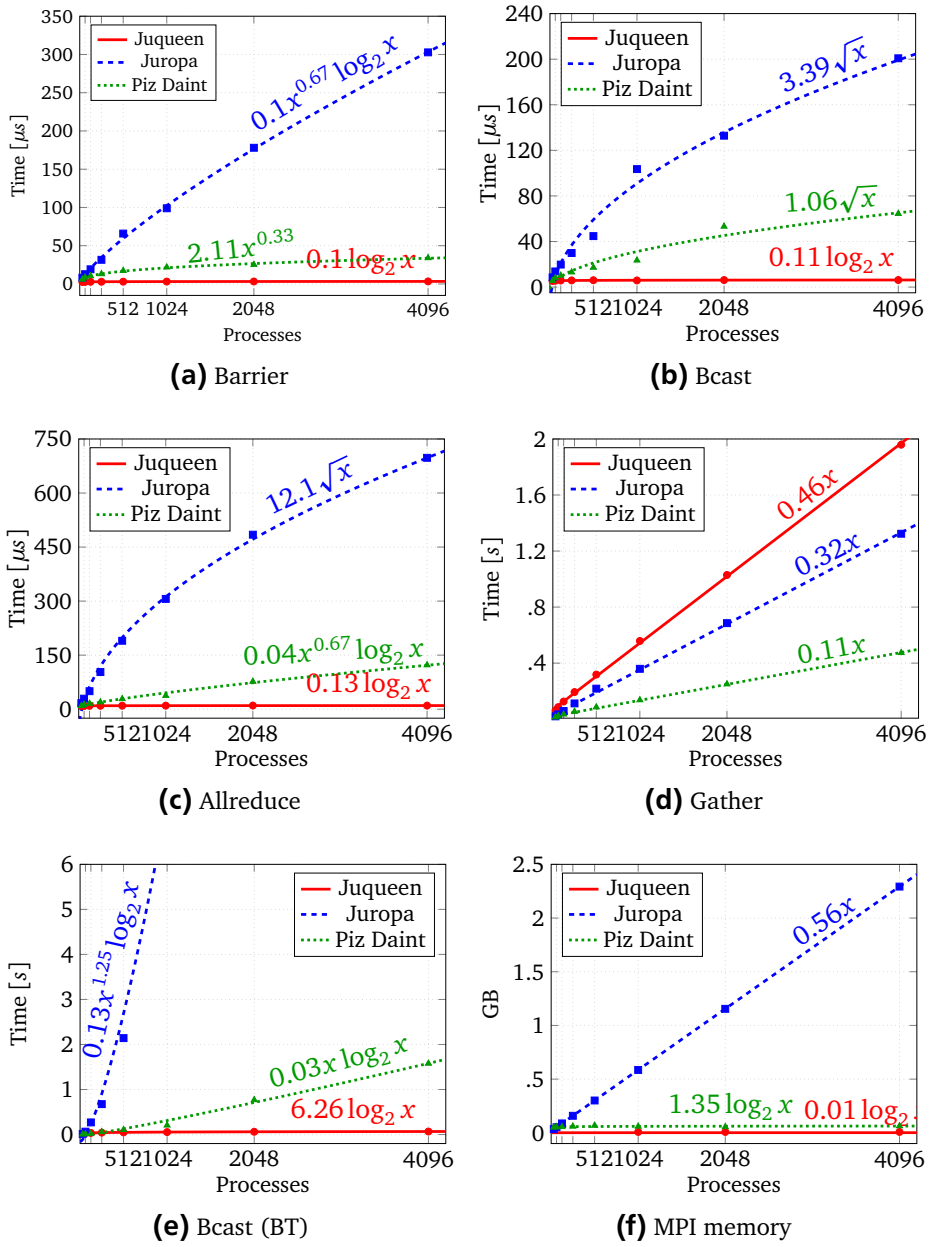


Figure 5.2: Measurements (circles, squares, triangles) and generated runtime models (plot lines) on Juqueen, Juropa, and Piz Daint [3].

Juqueen confirmed our expectations, while the MPI implementations on Juropa and Piz Daint both had performance issues.

5.1.3 Discussion

The scalability framework promises a streamlined solution to the performance modeling of HPC libraries and applications. It provides a ready-to-use method to understand the performance of any application and library, helps improve development techniques by including constant checks on performance and asks only a minimum of effort from the developers themselves.

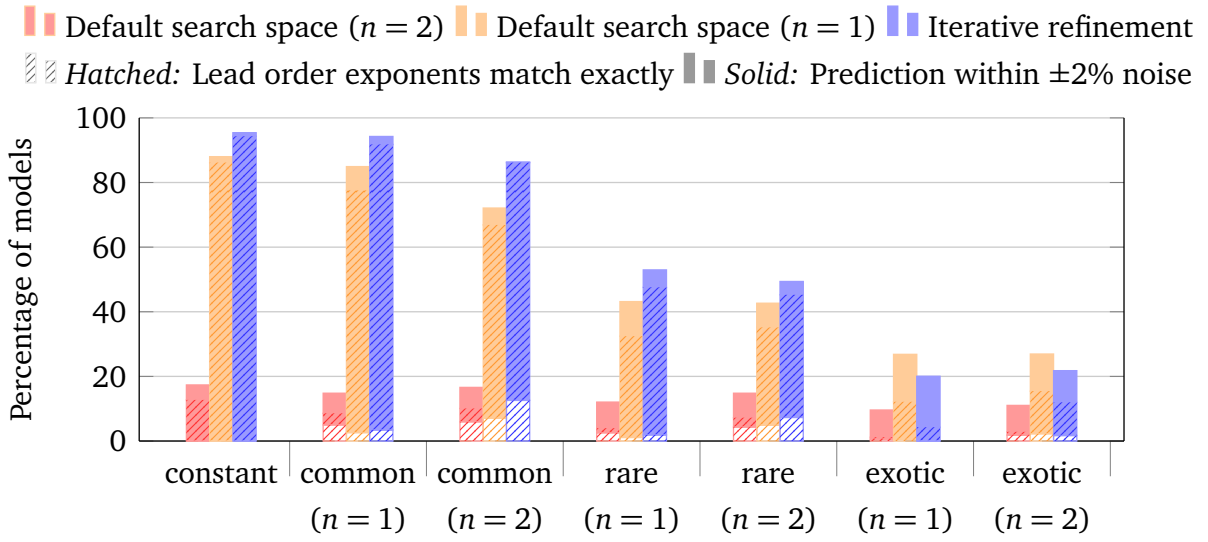


Figure 5.3: Comparison of the original and the new algorithm using values of randomly generated functions with ±2% of noise as input. The functions are built according to the PMNF with $n = 1$ or $n = 2$, and their coefficients c_0, c_1 and c_2 are calculated by sampling $a \in [-2, 3]$ uniformly and then computing 10^a [4].

5.2 Iterative refinement

The scalability framework first suggested creating ad-hoc search spaces around particular performance expectations, to allow for more detailed models to be generated. A recent improvement by Reisert et al. [4] utilizes this idea, but attempts to provide a faster and more accurate general way of creating performance models. The new approach applies the principles first established in the golden section search heuristic. The golden section search is introduced as a method to speed up the multi-parameter performance model search in Chapter 3.

In the original approach presented in Chapter 2, the search space was defined as a set of possible instantiations of the PMNF by choosing different values for the exponents of logarithms and polynomials forming each hypothesis. The new approach is still constrained by hypotheses that can be represented by the performance model normal form. The new approach starts from any hypothesis, for example a simple linear model, and examines additional hypotheses in its vicinity. It then selects the one which best fits the data and repeats the process. Therefore, the models can become arbitrarily accurate, without requiring a predefined search space. This unburdens the user who no longer has to provide expectations for code behavior. While such expectations can still be incorporated for example as starting points, this will affect the speed of convergence not the accuracy of the result.

This new approach therefore foregoes the requirement to define the search space entirely while still retaining the goal of obtaining as accurate a model as possible. Compared to the original approach, the iterative method provides better results for both synthetic data as well as all scientific applications tested. The evaluation with synthetic data, presented in Fig.5.3, follows the same steps defined in in Section 2.6. The evaluation shows that the iterative refine-

Table 5.1: Comparison of the original and the improved algorithm. Data from previous case studies is used. To show the quality of predictions, the last measured data point is not used for modeling but for comparing the resulting performance models [4].

Benchmark	Number of points	Model count	Model predictions (percentage of all models)			Mean relative prediction error [%]	
			better	same	worse	before	now
Sweep3D [24]	7	96	26.04	56.25	17.71	17.26	6.31
HOMME [24]	9	670	18.81	68.51	12.69	3.69	3.03
MILC [24]	9	1496	30.95	56.48	12.57	36.71	14.53
UG4 [44]	5	2026	52.62	38.01	9.38	68.30	15.58
MPI coll. [3]	7–8	26	65.38	7.69	26.92	52.53	15.89
BLAST [25]	5	103	31.07	41.75	27.18	34.92	10.38
Kripke [25]	5	36	36.11	38.89	25.00	33.05	8.32
Total	5–9	4453	39.12	49.11	11.77	45.71	12.97

ment approach is superior to the classic algorithm for tests using common and rare exponents for terms, and has only slightly worse results for exotic terms.

By applying the iterative refinement approach to the scientific applications previously analyzed, the improvement in accuracy obtained is displayed in Table 5.1. The number of accurate predictions is greater and the mean relative prediction error is smaller for the new approach, reinforcing the conclusion that this is a significant improvement not only for the usability of the tool but rather for its accuracy as well. This powerful improvement has become a new feature of Extra-P and will be included in the next release of the tool as the default way of generating performance models.

5.3 Isoefficiency

Determining the correct problem decomposition and computational load to achieve maximum performance in task-based programming is a difficult task and usually one that requires significant experimentation. Shudler et al.[9] first create performance expectations by analyzing the task dependency graph of an application. Then, they analyze the efficiency of task-based applications with the help of the powerful empirical multi-parameter modeling made possible by Extra-P. By combining these two approaches they determine the isoefficiency of task-based applications. This enables us to answer questions like “What is the required core count for a given input size such that a given efficiency is maintained?”

Multi-parameter modeling is used to create isoefficiency models, by computing the efficiency as a function of problem size and thread count. Table 5.2 shows the efficiency models determined for a number of different algorithm types. For each algorithm, the first row indicates the actual efficiency measured empirically, while the second row indicates the theoretical effi-

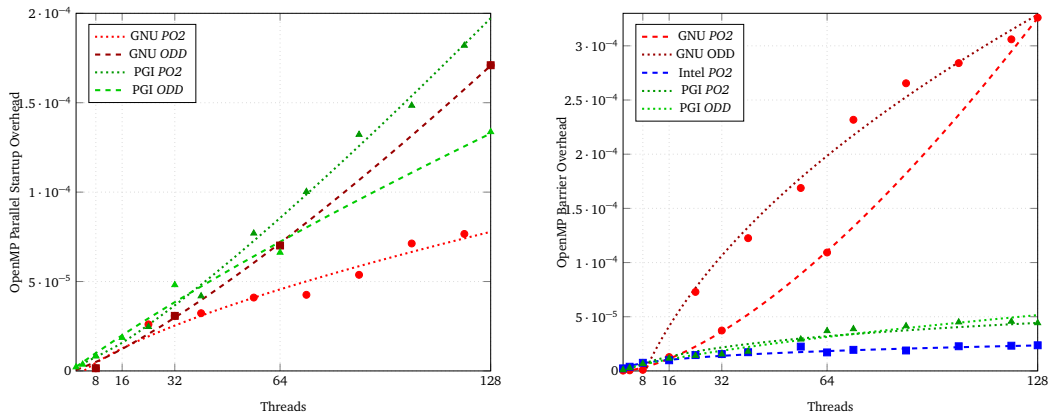
Table 5.2: Efficiency models of applications selected form the Barcelona OpenMP Task Suite [8]. The last column shows the required input sizes (n) for $p = 60$ and an efficiency of 0.8 [9].

Application	Model		rRMSE	Input size for $p = 60$
Cholesky	E_{ac}	$1.09 - 0.51 \cdot \sqrt{p} + 3.11 \cdot 10^{-2} \cdot \sqrt{p} \log n$	9.7%	37,718 × 37,718
	E_{cf}	$1.14 - 0.54 \cdot \sqrt{p} + 3.4 \cdot 10^{-2} \cdot \sqrt{p} \log n$	7.8%	24,685 × 24,685
FFT	E_{ac}	$0.96 - 0.1 \cdot \log p + 5.08 \cdot 10^{-22} n^{4.5} \log p$	19.5%	30,310 × 30,310
	E_{cf}	$1.03 - 0.16 \cdot p^{0.67} + 1.04 \cdot 10^{-2} \cdot p^{0.67} \log n$	4.8%	15,800 × 15,800
Fibonacci	E_{ac}	$0.98 - 5.11 \cdot 10^{-3} \cdot p^{1.25} + 1.76 \cdot 10^{-3} \cdot p^{1.25} \log n$	3.5%	51
	E_{cf}	$0.97 - 1.46 \cdot 10^{-2} \cdot p^{1.25} + 9.26 \cdot 10^{-3} \cdot p^{1.25} \log n$	3.0%	51
NQueens	E_{ac}	$1.04 - 0.66 \cdot \sqrt{p} + 0.17 \cdot \sqrt{p} \log n$	13%	14
	E_{cf}	$1.0 - 6.21 \cdot 10^{-2} \cdot p + 1.61 \cdot 10^{-2} \cdot p \log n$	3%	13
Sort	E_{ac}	$0.99 - 9.2 \cdot 10^{-3} \cdot p^{1.5} + 2.29 \cdot 10^{-4} \cdot p^{1.5} \log n$	1.9%	350G
	E_{cf}	$1.0 - 4.61 \cdot 10^{-2} \cdot p^{0.75} + 1.62 \cdot 10^{-3} \cdot p^{0.75} \log n$	5.7%	6.6M
SparseLU	E_{ac}	$1.02 - 0.46 \cdot p^{0.67} + 3.28 \cdot 10^{-2} \cdot p^{0.67} \log n$	6.3%	12,000 × 12,000
	E_{cf}	$1.05 - 0.48 \cdot p^{0.67} + 3.49 \cdot 10^{-2} \cdot p^{0.67} \log n$	6.1%	11,000 × 11,000
Strassen	E_{ac}	$1.55 - 1.02 \cdot p^{0.25} + 4.59 \cdot 10^{-2} \cdot p^{0.25} \log n$	9.5%	83,600 × 83,600
	E_{cf}	$1.26 - 0.65 \cdot p^{0.33} + 3.89 \cdot 10^{-2} \cdot p^{0.33} \log n$	5.9%	12,680 × 12,680

ciency that could be obtained in a contention-free environment. The contention-free values are determined by replaying the algorithm run in a simulated contention-free environment.

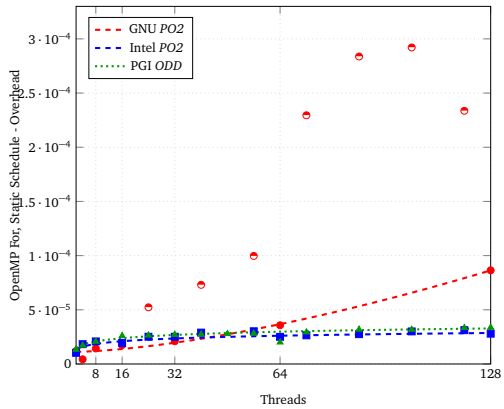
The relative residual mean squared error (rRMSE) is quite small for most examples, being larger than 10% in only 2 out of 14 cases. What we can observe is that although the exponents may differ, all models have a similar structure. This is explained, and expected by the behavior which is modeled through efficiency: We start with a constant efficiency around one. As more processes or threads are added, the efficiency shrinks, therefore the second term is negative, and always a function of p, the number of threads. As the problem size n, is increased, some of the inefficiency added by the extra threads is mitigated. That explains the third, positive term which is a function of both the thread count and problem size. The coefficients are much smaller for this term. While this behavior was obvious in hindsight, the authors of the study did not assume this at the start of their work and the models obtained using Extra-P helped understand the behavior of task-based programs and acted as an additional sanity check and confirmed expectations.

An important aspect related to this work is that it highlights a limitation of the empirical modeling approach. The empirical nature of the models does not capture or predict corner case behaviors well. For example, the efficiency can never grow to surpass one, but in the models we compute, there always exists a problem size n such that the resulting efficiency becomes greater than one. If we keep this limitation in mind however, Extra-P allows iso-efficiency models to be easily determined, and can serve as a powerful tool to balance efficiency with concurrency for task-based programs.

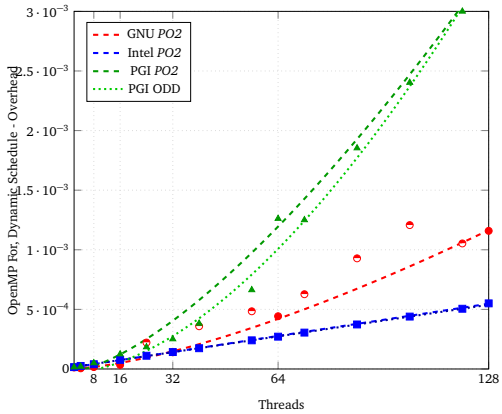


(a) BCS “parallel”

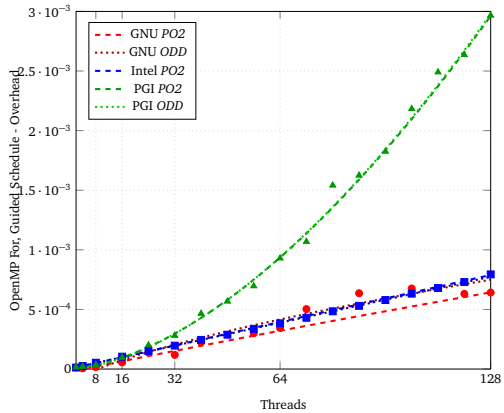
(b) BCS “barrier”



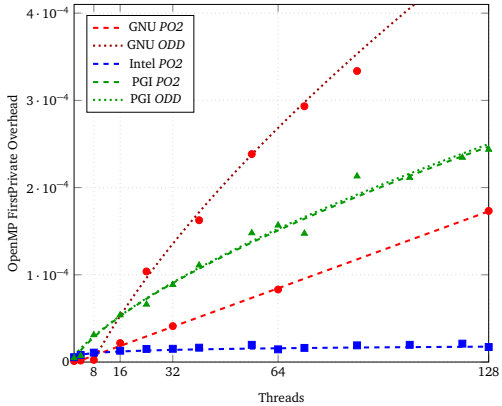
(c) BCS “static”



(d) BCS “dynamic”



(e) BCS “guided”



(f) BCS “first-private”

Figure 5.4: Measurement points used, denoted by circles, squares, and triangles, and model plots in dotted lines for OpenMP constructs using RWTH Aachen BCS Cluster for the compilers GNU 4.9, Intel 15 and PGI 14 [5].

5.4 OpenMP scalability study

Iwainsky et al.[5] used Extra-P to model the scalability of different OpenMP implementations. The authors developed a complex benchmarking system to gather measurements from four

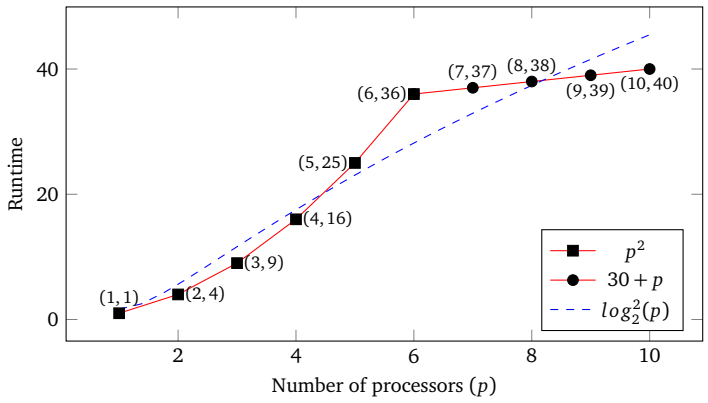


Figure 5.5: Data points from two different functions (solid lines) and the model generated by Extra-P (dashed line) [6].

different OpenMP implementation (GNU, IBM, Intel, and PGI) on three different hardware platforms (Xeon, Xeon Phi, and Blue Gene/Q). They then used Extra-P to create performance models, uncovering potentially serious scalability limitations for three of the four implementations, as well as proving through subsequent analysis that the behavior varies qualitatively depending on the number of threads being a power of two or not.

Figure 5.4 shows both the different behaviors observed for the different compilers, as well as the different models obtained for powers of two as opposed to odd numbers of threads. That different performance models can be obtained when looking at thread counts which are a power of two compared to when that is not the case seems obvious in hindsight. However, we observed that these models are not always better, as exemplified by the GNU barrier implementation: while the "odd" or not power of two data measurements start off taking a longer time, the growth rate is smaller than that of the model representing only powers of two. This was completely unexpected, and indicated a serious issue with that implementation.

5.5 Segmented performance modeling

While Extra-P is an effective tool in uncovering performance bottlenecks, it did suffer from an important limitation: Extra-P assumes that the performance of a kernel can be characterized by a single function. However, some kernels do not display a single behavior in every situation. In some situations, programs change their behavior. For example, modern MPI implementations switch from one algorithm to another, depending on the message size, the number of processes, or the network topology [62]. By ignoring the possibility that the input data represents two or more distinct behaviors, inaccurate models can be generated, scalability bugs can be ignored, or the user might encounter false positives.

Ilyas et al. [6] have developed a method for automatically detecting both segmented behavior as well as determining the interval where the behavior changes. They have tested this empirical approach with more than 5.2 million synthetic tests as well as three different application case studies.

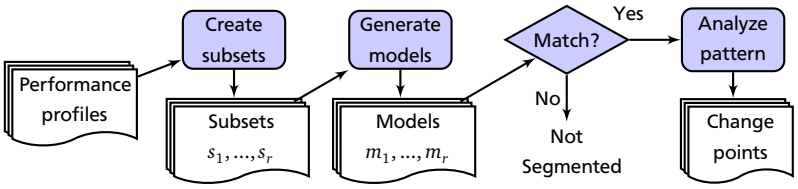


Figure 5.6: Steps involved in segmentation detection and change-point identification [6].

Figure 5.5 shows an example of segmented behavior. The first five points represent one behavior, while the last four represent another. The sixth point is common to both behaviors. The model discovered by the standard Extra-P approach is misleading as it represents neither behavior, and even worse, assumes that the asymptotic growth rate is worse than it actually is.

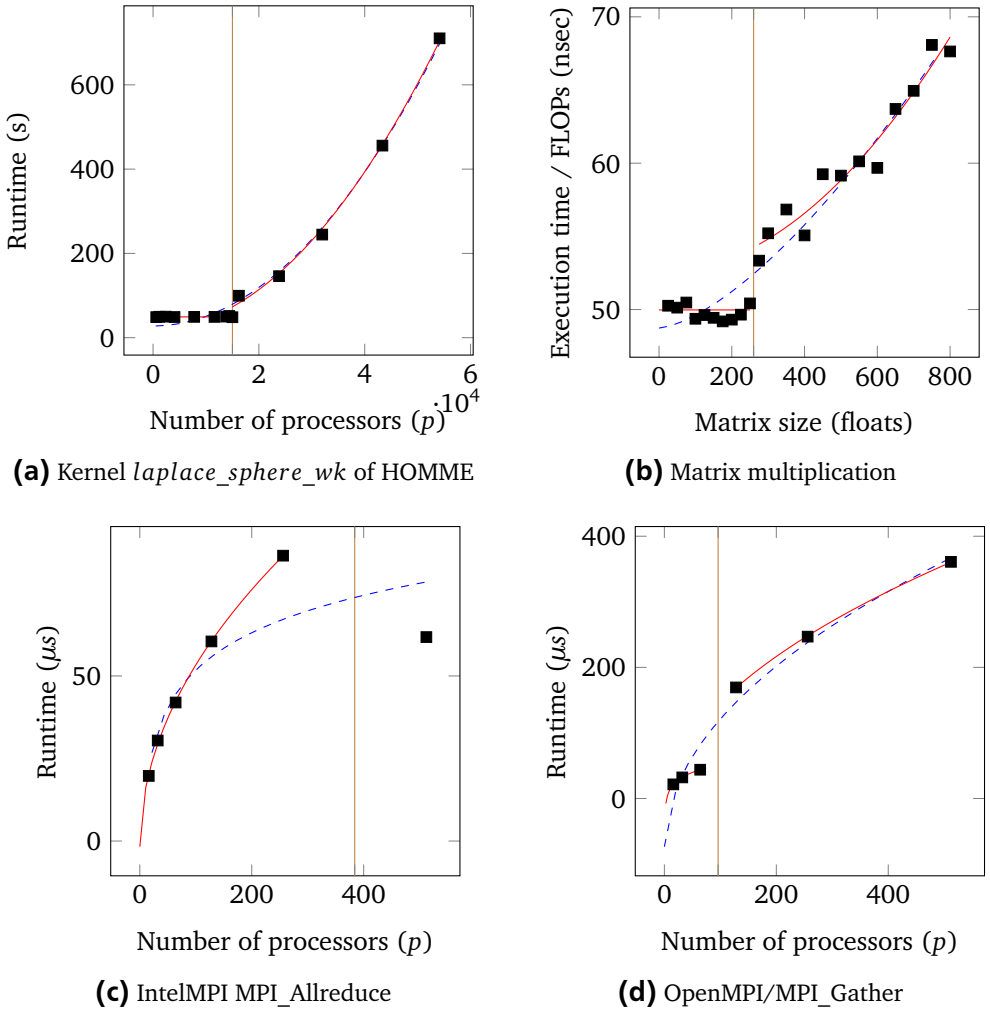
Figure 5.6 shows how the original Extra-P workflow is changed to allow for the discovery of segmented models. The idea is to separate the available data points into smaller subsets, model the subsets individually and then analyze these models. Using the same data as Figure 5.5, Table 5.3 shows the models for the different subsets as well as the metric identified by the authors as most helpful in deciding whether segmentation exists: the normalized residual sum of squares (nRSS). The authors determined empirically that this metric has high values for models created from measurements stemming from different behaviors. They also observed that the error for models containing measurements from two behaviors is usually multiple orders of magnitude higher than that of models containing measurements of only one behavior. This lead to the secondary criterium ϵ, defined as the quotient between different nRSS values.

The results of testing this method of detecting segmented behavior on real applications are displayed in Figure 5.7. A known issue of HOMME, a climate modeling application, discovered by Calotoiu et al. [24], was correctly identified. The formula by which the number of iterations a function performs is computed contains a *ceiling* term that limits the number of iterations to one for up to and including 15k processes. Beyond this threshold, a term depending quadratically on the process count causes the number of iterations executed to grow rapidly, causing a significant drop in performance. Figure 5.7a shows the constant behavior up to 15k processes and the quadratic behaviors afterwards.

The change in performance when the data for a matrix-matrix multiplication no longer fits in the cache was also identified and is displayed in Figure 5.7b. Finally, changes in the algo-

Table 5.3: Subsets created for the data from Figure 5.5, their respective models, and their nRSS values. Heterogeneous subsets are highlighted [6].

Subset	Model	nRSS	ϵ
$s_1 = \{1, 4, 9, 16, 25\}$	p^2	≈ 0	–
$s_2 = \{4, 9, 16, 25, 36\}$	p^2	≈ 0	≈ 1
$s_3 = \{9, 16, 25, 36, 37\}$	$-49.41 + 33.45 \cdot \sqrt{p}$	0.18	$5 \cdot 10^{18}$
$s_4 = \{16, 25, 36, 37, 38\}$	$-28.53 + 23.17 \cdot log_2(p)$	0.19	1.05
$s_5 = \{25, 36, 37, 38, 39\}$	$-6.19 + 14.83 \cdot log_2(p)$	0.16	0.84
$s_6 = \{36, 37, 38, 39, 40\}$	$30 + p$	≈ 0	≈ 0



(a) Kernel *laplace_sphere_wk* of HOMME

(b) Matrix multiplication

(c) IntelMPI MPI_Allreduce

(d) OpenMPI/MPI_Gather

Figure 5.7: Graphs showing measurement points, non-segmented models (dashed lines) and segmented models (solid lines). Estimated locations of change points are shown by vertical lines [6].

rithms used by MPI collective operations were also correctly discovered by this approach. Their existence was known and documented [62], and was used to verify that the approach works as intended. Figures 5.7c and 5.7d show that the segmentation due to the algorithm change is correctly identified. This improvement to Extra-P will broaden the range of behaviors that can be correctly identified, as well as eliminate a potential source for false positives and therefore will make the tool more powerful and versatile.

5.6 Discussion

Extra-P and the methods described in this work have allowed researchers to both better understand scientific codes as well as develop new performance analysis methods and workflows by creating and analyzing empirical automated performance models. While the automated performance modeling described in this work can, and will be improved, we believe it can already provide insight into performance questions for many applications. The areas of applicability

have so far been narrow, only focusing on HPC computing, but the potential of Extra-P to support performance experts and developers alike is greater in scope than that.

6 Extra-P — An Automatic Performance Modeling Tool

The methods described in this work have been implemented and released as open-source software, in the form of the performance modeling tool Extra-P [1]. The tool is freely available at and even though it has only been officially released in November 2015 it already sees use by different research groups such as the Lawrence Livermore National Laboratory, TU Darmstadt, the Jülich Research Center and the Goethe University in Frankfurt. To help developers familiarize themselves with Extra-P and its capabilities a number of tutorials has been held at research centers (Jülich), universities (RWTH Aachen, University of Mainz), and at HPC conferences such as SC and EuroMPI.

6.1 Performance modeling library

To allow developers flexibility in how they deploy and use Extra-P, the entire modeling functions, objects and methods are bundled in a library written in C++. The library itself has no dependencies and should therefore be easy to build and employ on every system. Extra-P is particularly flexible as at its core, only a set of measurements in the form of pairs of parameter configurations and measurement values is required to create a model. The different modeling options and methods described in this thesis are implemented in an object-oriented paradigm to allow for extensions to be easily added.

The creation of multiple performance models in parallel would be trivial to implement. However, at this time the library is able to create performance models for even the largest applications analyzed so far with multiple parameters in minutes or less, making parallelization efforts a low priority at this time.

The current release accepts performance profiles in the CUBE format generated by Score-P as an input, as well as measurement data in a simplified text-based format, which allows for fast model generation and debugging. Support for CSV and HDF5 formats is already in development and should become available in the next release. Due to the flexibility of Extra-P, further formats can, and have been developed ad-hoc for specific case studies where the available methods were unsuitable.

6.2 Graphical user interface

The first version of Extra-P was released as both a command-line tool and a plugin for CUBE [63], a tool for visualizing multi-dimensional performance spaces commonly used as

[1] http://www.scalasca.org/software/extra-p/download.html

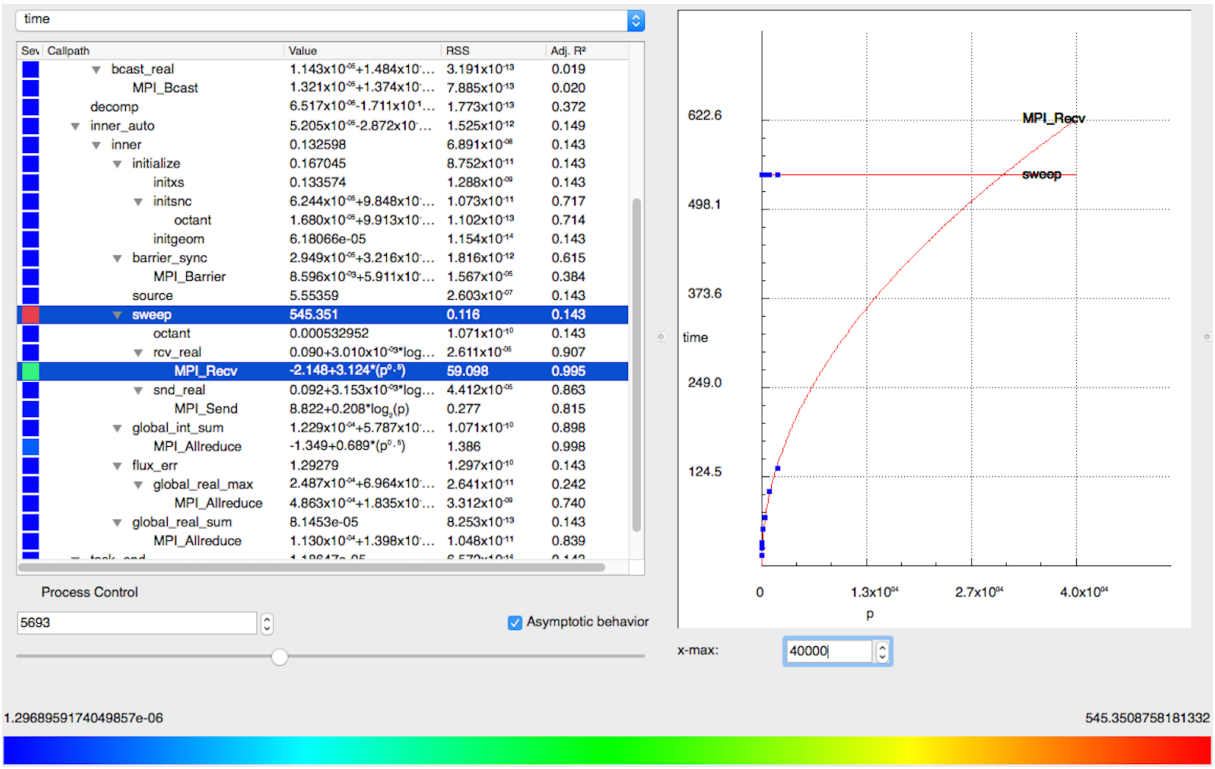


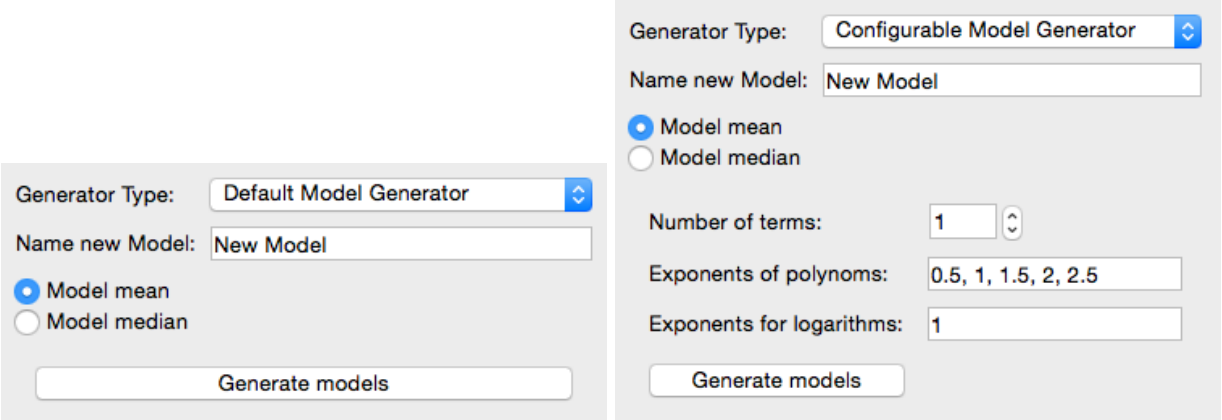
Figure 6.1: Interactive exploration of performance models with Extra-P. The screen shot shows performance models generated for different call paths in SWEEP3D, a neutron transport solver. The call tree on the left allows the selection of models to be plotted on the right. The color of the squares in front of each call path highlights the runtime at a user-selected process count. The call path ending in `MPI_RECV` has a measured complexity of $\sqrt{p}$.

a performance report explorer for Scalasca and Score-P. The current version has been developed as a standalone PyQT tool, to allow for more customization as well as a more streamlined design. Users can visualize and navigate the models generated with Extra-P with a simple but powerful GUI. Figure 6.1 shows an example.

Beyond performance analysis with the help of performance models, Extra-P allows users to also create performance models choosing from a selection of multiple model generators and even tune the modeling process by having access to the full suite of configuration options available in the Extra-P library. Two different performance model generators are displayed in Figure 6.2a and Figure 6.2b.

The first one, shown in Figure 6.2a, is the default model generator. It is the one described in Section 5.2 and allows for detailed models to be obtained without any expectations, while at the same time preventing over-fitting.

The second model generator, shown in Figure 6.2b, allows the user to redefine which terms are used in the search space definitions as well as how many terms are allowed in a performance model. This approach allows for a great deal of customization and is useful if a user has clear expectations of how an application performs and would like to test their assumptions.



(a) Default model generator **(b)** Refining model generator

Figure 6.2: Interactive generation of performance models with Extra-P.

6.3 Tutorials

Extra-P has been presented through tutorials as part of the EuroMPI 2015 conference, and at the SC15 and SC16 conferences.Furthermore Extra-P tutorials were included in two week-long application tuning workshops organized by VI-HPS [2], a virtual institute with members from many organizations and research groups developing performance analysis tools. As part of the first workshop organized by the EPE project [3], Extra-P was introduced and taught to members of research groups in Hessen and Rhineland-Palatinate.

The first part of the tutorial explains the theoretical foundations. The second part offers a live demonstration of the tool and its capabilities. The participants, using their own notebook computers and provided with a release version of Extra-P, are able to follow the demonstration and create insightful models for a number of examples.

To facilitate the demo session, the tool, incl. instructions, is available online for attendees [4]. Slides from past tutorials can also be downloaded [5]. We provide guidelines for attendees wishing to bring and model their own programs. The content of this tutorial is aimed at participants with knowledge of parallel computing but does not require prior knowledge of performance modeling.

Beyond helping users familiarize themselves with the different features of Extra-P the tutorials have been tremendously helpful in understanding how users employ Extra-P and which features they consider important. For example, the current interface has been optimized to both maximize the screen area for the common case of analyzing parallel scalability, while options for changing how performance models are generated can be found in easily accessible unobtrusive menus. This leads to a streamlined, simplified user experience, while allowing experts to still have precise control over the modeling process.

[2] http://www.vi-hps.org/
[3] http://www.sc.informatik.tu-darmstadt.de/res/pro/epe/epe_overview/index.de.jsp
[4] http://www.scalasca.org/software/extra-p/download.html
[5] http://www.scalasca.org/software/extra-p/documentation.html

6.4 Discussion

By building on top of established instrumentation frameworks such as Score-P, and providing an easy to use interface and streamlined workflow, Extra-P attempts to make performance modeling available to developers and performance engineers alike, and allow them to find performance bottlenecks faster and with less effort. Development of Extra-P is continuing and we intend to add features, improve the modeling performance and increase our teaching efforts to make it available to an ever increasing number of developers.

7 Related Work

This section reviews the previous research related to the contributions of this thesis in the context of the larger field of performance analysis. Relevant past work related to analytical modeling as well as other automatic approaches for performance modeling are discussed.

7.1 Performance analysis

Performance analysis and prediction of real-world application workloads is most important in high-performance computing. The methods presented in this thesis pertain to this domain, and therefore we present a number of established tools used for performance analysis in HPC.

Several performance tools, such as HPCToolkit [11], Scalasca [64], TAU [12], and Vampir [65], allow the programmer to observe the performance of real-world applications at impressive scales. Yet, those tools are limited to observations on a target system and at a given scale and cannot obtain insights outside the measurement range.

To make the development of our tool as cost-efficient as possible, it relies on a standard performance-measurement infrastructure. Specifically, it has been designed as an extension of Scalasca [64] and Score-P [13], which are well-established open-source toolsets that support the performance optimization of parallel programs by measuring and analyzing their runtime behavior. While Scalasca analyzes the performance using one performance measurement where it attempts to find communication and synchronization inefficiencies, Extra-P extends the analysis to sets of measurements generated using the Score-P infrastructure to create performance models.

7.2 Analyical modeling

Analytical performance modeling techniques have been used to model the performance of numerous important applications manually [20, 21]. It is well understood that analytical models have the potential of providing important insights into complex behaviors [66]. Performance models also offer insight into different parts of the system. For example, Boyd et al. used performance models to assess the quality of a tool chain, such as a compiler or runtime system [67]. A very important motivation for the use of performance models was presented by Petrini et al. [68]. In their study, the difference between actual and predicted performance led to the discovery of system noise as the source of seriously degraded performance. In general, there is consensus that performance modeling is a powerful tool for assessing an application's resource consumption and scalability.

Hoefler et al. aimed to further popularize performance modeling by defining a simple six-step process to create application performance models [1]. The described method leads to insight

into application scaling behavior but is tedious to apply to real codes. Bauer, Gottlieb, and Hoefler show how to model performance variations in this framework using simple statistical tools [22]. They also describe how to measure the influence of certain system parameters such as the network topology.

7.3 Automatic performance modeling

This Section focuses on approaches that generate performance models automatically rather than manually, even if that moves the focus away from human-readable general-purpose models but rather on models generated for a very specific purpose. These approaches vary the number of processes to determine the scalability of applications, often imposing restrictions on either architectures or applications.

For example, Ipek et al. propose multilayer artificial neural networks to *learn* application performance [69] and Lee et al. compare a set of different schemes for automated machine-based performance learning and prediction [70]. Zhai, Chen, and Zheng extrapolate single-node performance to complex parallel machines [71]. Wu and Müller [72] extrapolate traces to larger process counts and can thus predict communication operations. Their extrapolation relies on a trace compression scheme that assumes regular communications. Our method is based on lightweight profiles which can be generated without making prior assumptions. All these schemes aim to deliver the most accurate prediction but do not try to find a human-readable scaling function, thus limiting insight.

Coarfa et al. automatically compare pairs of measurements at different scales to identify scalability bottlenecks [73], whereas our approach creates explicit predictive models that describe the scaling behavior beyond the range of measurements. Barnes et al. use regression analysis to predict the scalability of applications [74] and is probably the work most similar to our own. The main differences are that they aim to predict the optimal number of CPUs to solve a certain problem, similar to the approach of Shudler et al. [9] we discuss in Section 5.3 while we are most interested in modeling both the execution time consumed for a specific run and the requirements for different resources such as FLOPS or network communication. For this, their tool considers is focused on strong scaling of the whole application while we can analyze both strong and weak scaling, and identify performance behaviors at the granularity of functions calls.

7.4 Multi-parameter performance modeling

We now look at approaches which consider the effect of multiple configuration parameters on the performance of applications. These methods vary in their degree of automation, from source code annotations to fully automated approaches that create performance models from empirical measurements.

PALM [75] supports the creation of true multi-parameter models but requires the user to annotate the source code with micro-models that apply only to small sections of the code. Following extensive and detailed per-function measurements, the underlying framework then

automatically combines these micro-models into structural macro-models. The approach is related to Aspen [76], a dedicated language to specify such micro-models. Our approach provides a higher degree of automation without prior source-code annotation. Vuduc et al. select the best implementation of a given algorithm by automatically generating a large number of candidates and then choosing the one that offers the best performance according to an empirically derived model — potentially with multiple parameters [77]. However, automatic is only the linear regression to determine the model coefficients. The model hypothesis itself must be chosen manually, which is why their approach, as far as performance modeling is concerned, cannot be considered truly automatic.

Finally, there are also automated methods for multi-parameter performance modeling. For example, Siegmund et al. analyze the interaction of different configuration options of an application and model how this affects the performance of the application as a whole [78]. The main difference is the supported diversity of model functions. While they allow only linear, quadratic or logarithmic functions, we allow a flexible combination of polynomials and logarithms. Furthermore, we apply heuristics to traverse the model search space more quickly, which is especially helpful in view of the higher model diversity we provide. Finally, we construct performance models for every function (with calling context) in an application - not just for the application as a whole. This allows optimization efforts to be channeled to where they will be most effective. Hoefler et al. generate multi-parameter performance models online [79]. Although already supported by prior static analysis, the online nature of their approach limits the size of the search space and thus the diversity of models quite significantly, and therefore adversely affects model accuracy. Finally, another multi-parameter approach was presented by Jayakumar et al. [80]. They extract execution signatures from their target applications, representing different execution phases, and match them with reference kernels stored in a database. If such a match exists, they use the performance model belonging to the reference kernel to predict execution times for varying numbers of processors and input sizes. If no match can be found for an execution phase, they apply static analysis to derive performance models. Model parameters include the core count p and only one input-size defining variable n at a time, which the user has to identify manually. The spectrum of model functions is quite small and restricted to $n, log(n), n/p, log(n)/p$.

7.5 Requirements engineering using performance modeling

We now investigate other approaches geared towards creating requirement models and supporting the co-design process. This is a complex challenge, comprising many different aspects, which explains the variations of the methods encountered.

Various groups utilize performance models to predict the performance of a code on different architectures. For example, Carrington et al. present a model-based framework for predicting application behavior on different computers [81], Marin and Mellor-Crummey utilize semi-automatically derived performance models to predict performance on different architectures [82], and Yang et al. predict application performance on different architectures by running

kernels on an available architecture and using the relative performance difference between the target and test architecture to guide their prediction [83].

Further, performance models have been used to determine node-level application requirements. For example, Carrington et al. use simple linear, logarithmic, exponential, or constant regression to determine application requirement scaling [84]. Our method goes beyond that by combining more complex functions leading to more accurate models while still retaining interpretability. In the widest sense, the roofline model [51, 85] can also be considered as an interpretable requirement model. It is, however, mostly designed as an optimization tool and thus not easily applicable to co-design.

Dongarra et al. presented a study on hypothetical exascale machines [86]. Gahvari and Gropp as well as Bhatele et al. used this study as a starting point and analyzed the theoretical feasibility of several computational algorithms on these machines. They show bounds on network requirements in terms of latency and bandwidth that would have to be satisfied in order to solve these problems [87, 88]. While extremely valuable, their studies are purely theoretical and not based on real applications. With our method, we enable similar studies for actual code bases.

Many co-design approaches rely on application and architecture simulation. Such simulations exist at numerous granularities, ranging from cycle-accurate [15, 16] to coarse model-driven [89], allowing to trade off accuracy for speed and feasibility. Detailed network behavior can be modeled using trace-driven simulations such as SimGrid [17], DIMEMAS [18], or PSINS [19]. Other tools such as BigSim [90], Silas [91], and MPI-SIM [92] complement simulations with the use of direct or kernel execution to assess computation and communication times more accurately. Such direct execution approaches often show severe memory limitations [93], especially in the exascale range. The main drawback of such simulations are the enormous resources required to run them. Even the least accurate simulations rarely scale to exascale workloads. Furthermore, simulation results provide little insight into the application scaling on their own (without additional human interpretation). Our requirement models need no resources beyond the small scale measurements required to produce them, enabling extreme-scale predictions at very low cost. Moreover, they are intuitive in that they allow direct statements such as “the required network bandwidth grows logarithmically with the system size”. This makes them very powerful even if the exact architecture of an exascale system is not available to a system designer.

8 Conclusion

The main goal of this thesis has been the development of an accessible performance modeling solution to allow application developers and users to more deeply understand the behavior of their codes. We provide a handy tool to get an overview of the performance issues in general and scalability issues in particular of a application. Furthermore, we attempt to pinpoint the issues and anchor them at a fine granularity in the code, thus offering a starting point for the debugging process. Our efforts do not replace the need for performance engineers and experts in analytical models, but rather allow them to focus on those kernels which prove to contain issues. In the following, the main contributions of this thesis are briefly summarized.

8.1 Automatic performance modeling

The first contribution of this work was to confirm that automatic performance modeling is feasible and that the automatically generated models are accurate enough to identify scalability bugs by introducing the performance model normal form and the fine-grained modeling approach. In fact, in those cases where hand-crafted models existed in the literature we found our models to be competitive. The main lesson that we learned during our work is that the advantages of mass production also apply to performance models. First, approximate models are acceptable as long as the effort to create them is low and they do not mislead the user. Second, code coverage is as important as model accuracy. Having approximate models for all parts of the code can be more useful than having a model with 100% accuracy for just a tiny portion of the code or no model at all. Extending this argument beyond the boundaries of a single application, we believe that our tool makes scalability modeling accessible to a much wider audience of HPC developers and applications.

Our tool models only behavior found in the training data. We provide direct feedback information regarding the number of runs required to ensure statistical significance of the modeling process itself, but there is no automatic way of determining at what scale particular behaviors start manifesting themselves. In the HOMME example, the iteration count suddenly increased after 15k processes, which was only detectable through either code analysis or experiments. We expect that our method will be most effective for regular problems with repetitive behavior.

8.2 Multi-parameter performance modeling

The method previously introduced focuses on the effect of one configuration parameter, usually the number of processes, has on performance. The second contribution of this work is the demonstration that automatic performance modeling with multiple parameters is feasible. Within seconds, we generated accurate performance models for realistic applications from a

limited set of performance measurements. The models both confirmed assumption the developers had earlier, providing further validation for our heuristics, and offered new unexpected insights into application behavior. Given that the resources needed for the model generation itself are now negligible, the number of model parameters is only constrained by the amount of measurements a user can afford. From a practical perspective, we believe that this will allow the key parameters of many applications to be captured in empirical performance models.

Speaking in general terms, our method enables a more complete traversal of the performance space compared to other performance analysis methods at relatively low cost, making application performance tuning more effective. Auto-tuning methods that still rely on long series of performance tests, which are not only time consuming but also expensive, can profit as well.

8.3 Requirements engineering using performance modeling

The third contribution of this work is an application of the multi-parameter performance modeling method. We introduce of a quick and simple automatic back-of-the-envelope technique to generate requirement models for parallel applications. This will not only benefit application developers but also the designers of emerging systems, who can now project application requirements more precisely along several parameter dimensions and balance their systems accordingly. Our tool generates interpretable models that can be used to gain quick insights into various aspects such as the required number of floating-point operations or network communication volumes to solve a particular problem with a particular number of processes. An important contribution is the ability to compare how the requirements for different resources change when a code is scaled. We demonstrate our lightweight modeling with five different applications. The requirement models enable system designers to consider various upgrade and design options. Our analysis demonstrates how the requirement models can be used to gain an intuitive understanding of critical system parameters and how to balance a system configuration to support a certain application. We believe that our method can be easily applied to a full compute center workload consisting of several dozens of applications.

8.4 Impact

The methods presented in this thesis already had an impact on the performance modeling community. From case studies of scientific applications such as UG4 [7] to implementations of parallel programming paradigms such as OpenMP [5] and MPI [3], Extra-P has already seen widespread use. Beyond its intended use as a tool to discover performance issues, it has been employed as a starting point to develop new performance analysis approaches altogether, such as determining the isoefficiency of task-based parallel programs [9].

The release of Extra-P as an open-source performance modeling tool ensures that the contributions of this thesis are not just theoretical, but rather immediately accessible and usable by any developer who wishes to gain performance insights from empirical measurements. Furthermore, we are actively teaching developers how to use our tool efficiently through workshops and

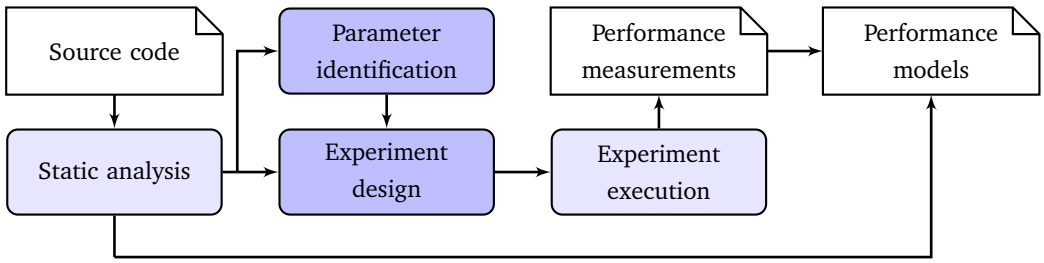


Figure 8.1: Compiler-assisted automatic performance modeling with multiple model parameters.

tutorials. In the future, as the performance modeling methods are refined and their capabilities bolstered through continuing research, Extra-P will continue to be improved.

8.5 Outlook

This section provides an outlook towards the different opportunities for continuing the research presented in this thesis. With Extra-P as a mature, tried, and tested tool for performance modeling of homogeneous parallel programs we are now turning our gaze towards applying the methods introduced in this thesis to other fields such as deep neural networks and graph algorithms, and tackling the specific issues rising from their particularities, as well as enriching Extra-P with additional capabilities.

8.5.1 Compiler-assisted multi-parameter performance modeling

Automating the generation of performance models for HPC applications with large and complex spaces of configuration parameters is a significant challenge. The approach presented in this thesis solves the problem of generating performance models that take into account the effect of multiple configuration parameters empirically from gathered measurements. The questions of how to select which configuration parameters to investigate and which ranges for their values provide most insight are still open. Furthermore, the number of measurements required to create performance models grows exponentially with the number of parameters considered.

Static analysis could be used to identify configuration parameters relevant for individual kernels and whether interactions between these parameters exist at the kernel level. This would simplify the design of experiments and at the same time it would reduce the time and effort spent gathering measurements. In turn, this would allow the performance implications of these parameters and their interplay to be studied more effectively.

The goal is to create an enhanced version of Extra-P, capable of efficiently generating performance models with multiple parameters individually for each target metric and call path of an application. Towards this purpose two methods must be developed:

- A method to identify those model parameters that are performance relevant enough to be included in the model.
- A method to choose the parameter assignments for the required performance experiments such that accurate models can be produced while keeping the cost of the experiments low.

Figure 8.1 illustrates the essential steps of this approach and highlights the major solution components (dark purple boxes). First, the solution needs to identify relevant model parameters, mainly using static analysis. Interactions are proved or disproved whenever possible. The next step is the design of the necessary performance experiments, that is, the selection of parameter assignments to be tried. The final step is the generation of the multi-parameter performance models.

We plan on building on the approach Hoefler et al. use to generate multi-parameter performance models online [94, 79]. The static analysis they use to guide the online modeling process will be expanded and used for parameter identification and experiment design.

8.5.2 Performance through decomposition

A key technique for understanding the behavior of complex systems is to decompose them into smaller parts and analyze them individually. For example, empirically determining the scalability models of parts of a parallel application with Extra-P can help determine the existence of bottlenecks.

However, having determined the scalability for parts of a parallel system is not sufficient to prove that these scalability properties hold for the entirety of the system. Deriving information about how performance effects are composed in parallel applications would be a powerful tool helping developers design new applications or improve and understand existing algorithms. By examining design patterns of parallel programs [95], and trying to identify properties to which the composition and decomposition could apply to, we could support the designing of parallel algorithms.

The challenge will be to define properties, propose assumptions under which the compositional reasoning is possible, and then test these assumptions using a concrete implementation of a design pattern. A study of scalability will require a concrete implementation to provide a starting point to validate or invalidate theoretical insights.

As an example, we propose a pipeline where each stage in the pipeline can process tasks in parallel. The different stages of the pipeline could be connected by unbounded queues. In this example, two of the questions to be answered is what properties need the tasks to have such that the compositional reasoning is possible? Are the performance models of the whole compared to the different parts additive or multiplicative or another arbitrary function?

8.5.3 Performance modeling of graph algorithms

Graph algorithms represent an important subclass of programs, which we have not attempted to create performance models for in the past. The challenge herein is that the behavior of graph algorithms depends very much on their input and in case of parallel algorithms on the problem decomposition.

Taking the entire structure of a graph into account and considering each edge and each node as a separate configuration parameter for the modeling process will lead to a untenably large search space. It is unlikely that enough measurements showing a variation for each combina-

tion of parameters could even be gathered. The challenge will therefore be to identify which configuration parameters describe the performance of graph algorithms. The study should investigate whether certain classes of graph algorithms such as searching, routing, coloring have common configuration parameters which affect their performance in similar ways.

For example, for some graph algorithms the number of nodes, the number of edges, the degree of a node, or the combination of some of the meta-parameters describing the graph could dominate its performance. This could allow models to be created from a reasonably small number of performance measurements.

8.5.4 Performance modeling and deep neural networks

Deep neural networks (DNNs) [96] have made machine learning a very relevant topic over the last few years. By applying deep neural networks in tasks such as speech recognition and image recognition results have been achieved that were previously considered impossible with traditional machine learning approaches. However, neural networks must be trained with the use of a large number of parameters (referring in this context to the adjustable weights of different inputs of the network) to reach desired outcomes. Training neural networks with billions of parameters as is required for many real applications, such as image or speech recognition, needs a lot of computation power.

Performance modeling with deep neural networks. A first aspect to pursue would be to generate empirical performance models using deep neural networks and compare the results with the approaches presented in this work both from the perspective of accuracy as well as time to solution. Susceptibility to noise will likely play a major role in this evaluation. An important challenge will be how to organize and provide the measurements to train the neural network.

Performance modeling of deep neural networks. The training time of deep neural networks and the effects of varying configuration parameters on it have been intensely studied [97, 98]. However, the performance effect of configuration parameters on the deployment of neural networks is still insufficiently understood. Obtaining measurements of deployed DNNs should be tractable given the existing frameworks, and allow Extra-P to determine multi-parameter performance models. Of particular interest would be the compounded effect the number of layers together with the size of each layer have on both performance and accuracy of the results.

8.6 Discussion

The most important contribution of this work is a powerful empirical method for creating performance models, as well as an open source tool to implement it. Many improvements both with respect to capabilities and accuracy have been added and we consider this approach to be fit for its purpose and capable of tackling most issues that come up in the modeling of homogeneous parallel applications. Looking forward, the next steps are to apply this approach to other

fields and other types of applications, as well as investigating alternative and complementary approaches to the empirical performance modeling method presented in this thesis.

Bibliography

[1] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, SC '11, pages 6:1–6:12. ACM, 2011.

[2] B. J. N. Wylie, M. Geimer, M. Nicolai, and M. Probst. Performance analysis and tuning of the XNS CFD solver on BlueGene/L. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface : Proceedings 14th European PVM/MPI Conference, EuroPVM/MPI'07, Paris, France - Berlin, Springer, 2007. - (Lecture Notes in Computer Science ; 4757). - 978-3-540-75416-9. - S. 107 - 116*, 2007. Record converted from VDB: 12.11.2012.

[3] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf. Exascaling your library: Will your implementation meet your expectations? In *Proc. of the International Conference on Supercomputing (ICS), Newport Beach, CA, USA*, pages 165–175. ACM, June 2015.

[4] P. Reisert, A. Calotoiu, S. Shudler, and F. Wolf. Following the blind seer – creating better performance models using less information. In *Proc. of the 23rd Euro-Par Conference, Santiago de Compostela, Spain*, Lecture Notes in Computer Science, pages 106–118. Springer, August 2017.

[5] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf. How many threads will be too many? on the scalability of openmp implementations. In *Proc. of the 21st Euro-Par Conference, Vienna, Austria*, volume 9233 of *Lecture Notes in Computer Science*, pages 451–463. Springer, August 2015.

[6] K. Ilyas, A. Calotoiu, and F. Wolf. Off-road performance modeling – how to deal with segmented data. In *Proc. of the 23rd Euro-Par Conference, Santiago de Compostela, Spain*, Lecture Notes in Computer Science, pages 36–48. Springer, August 2017.

[7] A. Vogel, S. Reiter, M. Rupp, A. Nägel, and G. Wittum. UG 4: A novel flexible software system for simulating PDE based models on high performance computers. *Comp. Vis. Sci.*, 16(4):165–179, 2013.

[8] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[9] S. Shudler, A. Calotoiu, T. Hoefler, and F. Wolf. Isoefficiency in practice: Configuring and understanding the performance of task-based applications. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Austin, TX, USA*, pages 131–143. ACM, February 2017.

[10] N. Bhatia, F. Song, F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Automatic experimental analysis of communication patterns in virtual topologies. In *Proc. of the International Conference on Parallel Processing (ICPP), Oslo, Norway*, pages 465–472. IEEE Society, June 2005.

[11] L. Adhianto, S. Banerjee, M. W. Fagan, M. W. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, April 2010.

[12] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[13] D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, pages 85–97. Gauß-Allianz, Springer, 2012.

[14] R. Miceli, A. Berariu, and M. Gerndt. Introduction to automatic tuning of hpc applications: The periscope tuning framework, 04 2015.

[15] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, Ch. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Performance Eval. Review*, 31:8–12, March 2004.

[16] A. F. Rodrigues, R. C. Murphy, P. Kogge, and K. D. Underwood. The structural simulation toolkit: exploring novel architectures. In *Proc. of the ACM/IEEE Conference on Supercomputing*, (SC '06). ACM, 2006.

[17] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a generic framework for large-scale distributed experiments. In *Proc. of the 10th Intl. Conference on Computer Modeling and Simulation*, (UKSIM), pages 126–131. IEEE Computer Society, 2008.

[18] V. Subotic, J. C. Sancho, J. Labarta, and M. Valero. A simulation framework to automatically analyze the communication-computation overlap in scientific applications. In *Proc. of the IEEE Conference on Cluster Computing*, (Cluster '10), pages 275–283. IEEE Computer Society, 2010.

[19] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely. PSINS: An open source event tracer and execution simulator for MPI applications. In *Proc. of the Euro-Par Conference*, pages 135–148. Springer-Verlag, 2009.

[20] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC'01)*, page 37. ACM, 2001.

[21] M. M. Mathis, N. M. Amato, and M. L. Adams. A general performance model for parallel sweeps on orthogonal grids for particle transport calculations. Technical report, College Station, TX, USA, 2000.

[22] G. Bauer, S. Gottlieb, and T. Hoefler. Performance modeling and comparative analysis of the MILC lattice QCD application su3_rmd. In *Proc. of CCGrid*, May 2012.

[23] W. Gropp and M. Snir. Programming for exascale computers. *Computing in Science Engineering*, 15(6):27–35, Nov 2013.

[24] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. Nov. 2013. IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13).

[25] A. Calotoiu, D. Beckingsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. Fast multi-parameter performance modeling. In *Proc. of the 2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan*, pages 172–181. IEEE Computer Society, September 2016.

[26] F. Durst and M. Schaefer. A parallel block-structured multigrid method for the prediction of incompressible flows. *International Journal for Numerical Methods in Fluids*, 22(6):549–565, 1996.

[27] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(3):12–21, 1993.

[28] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[29] R. R. Picard and R. D. Cook. Cross-validation of regression models. *Journal of the American Statistical Association*, 79(387):575–583, 1984.

[30] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC '10)*, November 2010.

[31] D. M. Hawkins, S. C. Basak, and D. Mills. Assessing model fit by cross-validation. *Journal of Chemical Information and Computer Sciences*, 43(2):579–586, 2003.

[32] A. Blum, A. Kalai, and J. Langford. Beating the hold-out: bounds for k-fold and progressive cross-validation. In *Proc. of the 12th Annual Conference on Computational Learning Theory*, (COLT), pages 203–208. ACM, 1999.

[33] P. Zhang. Model selection via multifold cross validation. *The Annals of Statistics*, 21(1):pp. 299–313, 1993.

[34] D. S. Carter. Comparison of different shrinkage formulas in estimating population multiple correlation coefficients. *Educational and Psychological Measurement*, 39(2):261–266, 1979.

[35] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[36] Message Passing Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[37] M. Harlacher, A. Calotoiu, J. Dennis, and F. Wolf. Analysing the scalability of climate codes using new features of scalasca. In Kurt Binder, Marcus Müller, Manfred Kremer, and Alexander Schnurpfeil, editors, *Proc. of the John von Neumann Institute for Computing (NIC) Symposium 2016, Juelich, Germany*, volume 48 of *NIC Series*, pages 343–352. Forschungszentrum Jülich, John von Neumann-Institut for Computing, February 2016.

[38] Los Alamos National Laboratory. ASCI SWEEP3D v2.2b: Three-dimensional discrete ordinates neutron transport benchmark, 1995.

[39] D. Böhme, M. Geimer, F. Wolf, and L. Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Proc. of the 39th Intl. Conference on Parallel Processing (ICPP), San Diego, CA, USA*, pages 90–100. IEEE Computer Society, September 2010.

[40] A. Hoisie, O. M. Lubeck, and H. J. Wasserman. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Workshop on Wide Area Networks and High Performance Computing*, pages 171–187. Springer-Verlag, 1999.

[41] H. Wasserman, A. Hoisie, and O. Lubeck. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The Intl. Journal of High Performance Computing Applications*, 14:330–346, 2000.

[42] B. J. N. Wylie, M. Geimer, B. Mohr, D. Boehme, Z. Szebenyi, and F. Wolf. Large-scale performance analysis of SWEEP3D with the Scalasca toolset. *Parallel Processing Letters*, 20(04):397–414, 2010.

[43] J. M. Dennis, J. Edwards, K. J. Evans, O. Guba, P. H. Lauritzen, A. A. Mirin, A. St-Cyr, M. A. Taylor, and P. H. Worley. CAM-SE: A scalable spectral element dynamical core for the community atmosphere model. *Intl. Journal of High Performance Computing Applications*, 26(1):74–89, 2012.

[44] A. Vogel, A. Calotoiu, A. Strube, S. Reiter, A. Nägel, F. Wolf, and G. Wittum. 10,000 performance models per minute - scalability of the ug4 simulation framework. In *Proc. of the 21st Euro-Par Conference, Vienna, Austria*, volume 9233 of *Lecture Notes in Computer Science*, pages 519–531. Springer, August 2015.

[45] A. J. Kunen. Kripke - user manual v1.0. Technical Report LLNL-SM-658558, Lawrence Livermore National Laboratory, August 2014.

[46] S. Langer, I. Karlin, V. Dobrev, M. Stowell, and M. Kumbera. Performance analysis and optimization for blast, a high order finite element hydro code. *Proceedings of the 2014 NECDC*, 2014.

[47] L. I. Sedov. Propagation of strong shock waves. *Journal of Applied Mathematics and Mechanics*, 10:241–250, 1946.

[48] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpeCL and CUDA. In *Proceedings of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation*, pages 465–471, November 2012.

[49] G. A. Sod. A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws. *Journal of Computational Physics*, 27(1):1–31, April 1978.

[50] T. S. Bailey and R. D. Falgout. Analysis of massively parallel discrete-ordinates transport sweep algorithms with collisions. In *International Conference on Mathematics, Computational Methods & Reactor Physics, Saratoga Springs, NY*, 2009.

[51] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[52] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[53] E. Berg. *Efficient and Flexible Characterization of Data Locality through Native Execution Sampling*. PhD thesis, Department of Information Technology, Uppsala University, November 2005.

[54] D. Eklov and E. Hagersten. Statstack: Efficient modeling of lru caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 55–65, March 2010.

[55] S. Byna, A. Uselton, Prabhat, D. Knaak, , and Y. He. Trillion Particles, 120,000 cores, and 350 TBs: Lessons Learned from a Hero I/O Run on Hopper. 2013.

[56] I Karlin, J Keasler, and JR Neely. Lulesh 2.0 updates and changes. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.

[57] S. Rinke, M. Butz-Ostendorf, M. Hermanns, M. Naveau, and F. Wolf. A scalable algorithm for simulating the structural plasticity of the brain. In *Proc. of the 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Los Angeles, CA, USA*, pages 1–8, October 2016.

[58] H. Jasak and A. Jemcov. Openfoam: A c++ library for complex physics simulations. In *International Workshop on Coupled Methods in Numerical Dynamics, IUC*, pages 1–20, 2007.

[59] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[60] U. Ghia, K. N. Ghia, and C. T. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48:387–411, 1982.

[61] P. van Phuc. Large Scale Transient CFD Simulations for Buildings using OpenFOAM on a World's Top-class Supercomputer. In *4th Annual OpenFOAM User Conference, Cologne, Germany*, Lecture Notes in Computer Science, 2016.

[62] H. Steve. *Intel* MPI library collective optimization on the Intel Xeon Phi coprocessor using environment variable collective operation control, 2015.

[63] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr. Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352, June 2015.

[64] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[65] W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[66] S. Pllana, I. Brandic, and S. Benkner. Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art. In *Proc. of the 1st Intl. Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 279–284, 2007.

[67] E. L. Boyd, W. Azeem, H. Lee, T. Shih, S. Hung, and E. S. Davidson. A hierarchical approach to modeling and improving the performance of scientific applications on the KSR1. In *Proc. of the Intl. Conference on Parallel Processing*, (ICPP), pages 188–192. IEEE Computer Society, 1994.

[68] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the ACM/IEEE Conference on Supercomputing*, (SC '03), page 55. ACM, 2003.

[69] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Proc. of the 11th Intl. Euro-Par Conference*, pages 196–205. Springer-Verlag, 2005.

[70] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP '07), pages 249–258. ACM, 2007.

[71] J. Zhai, W. Chen, and W. Zheng. PHANTOM: Predicting performance of parallel applications on large-scale parallel machines using a single node. *SIGPLAN Notices*, 45(5):305–314, January 2010.

[72] X. Wu and F. Müller. ScalaExtrap: Trace-based communication extrapolation for SPMD programs. *ACM Transactions on Programming Languages and Systems*, 34(1), April 2012.

[73] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of SPMD codes using expectations. In *Proc. of the 21st Intl. Conference on Supercomputing*, (ICS), pages 13–22. ACM, 2007.

[74] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *Proc. of the Intl. Conference on Supercomputing*, (ICS), pages 368–377. ACM, 2008.

[75] N. R. Tallent and A. Hoisie. Palm: Easing the burden of analytical performance modeling. In *Proc. of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 221–230, New York, NY, USA, 2014. ACM.

[76] K. L. Spafford and J. S. Vetter. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[77] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical Models for Empirical Search-Based Performance Tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94, February 2004.

[78] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 284–294, New York, NY, USA, 2015. ACM.

[79] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler. Using compiler techniques to improve automatic performance modeling. In *Proc. of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT'15)*, pages 1–12, San Francisco, CA, USA, 2015.

[80] A. Jayakumar, P. Murali, and S. Vadhiyar. Matching application signatures for performance predictions using a single execution. In *Proc. of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015)*, pages 1161–1170, May 2015.

[81] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, February 2006.

[82] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Performance Eval. Review*, 32(1):2–13, June 2004.

[83] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proc. of the ACM/IEEE Conference on Supercomputing*, (SC '05), page 40. IEEE Computer Society, 2005.

[84] L. Carrington, M. Laurenzano, and A. Tiwari. Characterizing large-scale hpc applications through trace extrapolation. *Parallel Processing Letters*, 23(4), 2013.

[85] B. Norris, W. Spear, and A. Malony. Performance analysis of applications in the context of architectural rooflines. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 345–348, New York, NY, USA, 2017. ACM.

[86] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J. Andre, D. Barkai, J. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, C. Xuebin, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Zhong J., Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[87] H. Gahvari and W. Gropp. An introductory exascale feasibility study for FFTs and multigrid. In *IPDPS*, pages 1–9. IEEE, 2010.

[88] A. Bhatele, P. Jetley, H. Gahvari, L. Wesolowski, W. D. Gropp, and L. V. Kalé. Architectural constraints to attain 1 exaflop/s for three scientific application classes. In *IPDPS*, pages 80–91. IEEE, 2011.

[89] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim: simulating large-scale applications in the LogGOPS model. In *Proc. of the 19th ACM Intl. Symposium on High Performance Distributed Computing*, (HPDC), pages 597–604. ACM, 2010.

[90] G. Zheng, G. Kakulapati, and L. V. Kalé. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *Proc. of the 18th Intl. Parallel and Distributed Processing Symposium (IPDPS)*, page 78, April 2004.

[91] M. Hermanns, M. Geimer, F. Wolf, and B. J. N. Wylie. Verifying causality between distant performance phenomena in large-scale MPI applications. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 78–84. IEEE Computer Society, February 2009.

[92] R. Bagrodia, E. Deelman, and T. Phan. Parallel simulation of large-scale parallel applications. *Intl. Journal of High Performance Computing Applications*, 15(1):3–12, 2001.

[93] C. Mei. A preliminary investigation of emulating applications that use petabytes of memory on petascale machines. Master's thesis, University of Illinois at Urbana-Champaign, 2007.

[94] A. Bhattacharyya and T. Hoefler. PEMOGEN: Automatic adaptive performance modeling during program runtime. In *Proc. of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*, Edmonton, Alberta, Canada, 2014.

[95] Z. U. Huda, A. Jannesari, and F. Wolf. Using template matching to infer parallel design patterns. *ACM Trans. Archit. Code Optim.*, 11(4):64:1–64:21, January 2015.

[96] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[97] I. H. Chung, T. N. Sainath, B. Ramabhadran, M. Picheny, J. Gunnels, V. Austel, U. Chauhari, and B. Kingsbury. Parallel deep neural network training for big data on blue gene/q. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1703–1714, June 2017.

[98] X. W. Chen and X. Lin. Big data deep learning: Challenges and perspectives. *IEEE Access*, 2:514–525, 2014.

Alexandru Calotoiu

Curriculum Vitae

☎ +49 6151 16 27756
✉ calotoiu@cs.tu-darmstadt.de

Personal data

Date of birth: 5. May 1985

Place of birth: Bucharest, Romania

Education

2017 **Ph.D. defended**, *Department of Computer Science*, Technische Universität Darmstadt, *Passed with distinction*.

2011 **M. Sc. Simulation Sciences**, *Faculty of Mechanical Engineering*, RWTH Aachen University.

2009 **Dipl. Ing**, *Department of Computer Science Engineering*, Politechnical University Bucharest, *Passed with distinction*.

Professional Experience

2015-present **Research Associate**, *Technische Universität Darmstadt, Laboratory for Parallel Programming*.

2014 **Scientific Research Intern**, *Lawrence Livermore National Laboratory*.

2011-2015 **Research Associate**, *RWTH Aachen University, German Research School for Simulation Sciences, Laboratory for Parallel Programming*.

2008 **Software Development Engineer Intern**, *Microsoft Corporation*.