Empirical Modeling of Spatially Diverging Performance

Alexandru Calotoiu
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
acalotoiu@inf.ethz.ch

Markus Geisenhofer
Dept. of Mech. Engineering
TU Darmstadt
Darmstadt, Germany
geisenhofer@fdy.tu-darmstadt.de

Florian Kummer
Dept. of Mech. Engineering
TU Darmstadt
Darmstadt, Germany
kummer@fdy.tu-darmstadt.de

Marcus Ritter
Dept. of Computer Science
TU Darmstadt
Darmstadt, Germany
ritter@cs.tu-darmstadt.de

Jens Weber
Dept. of Mech. Engineering
TU Darmstadt
Darmstadt, Germany
weber@fdy.tu-darmstadt.de

Torsten Hoefler
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Martin Oberlack
Dept. of Mech. Engineering
TU Darmstadt
Darmstadt, Germany
oberlack@fdy.tu-darmstadt.de

Felix Wolf
Dept. of Computer Science
TU Darmstadt
Darmstadt, Germany
wolf@cs.tu-darmstadt.de

***Abstract*—A common simplification made when modeling the performance of a parallel program is the assumption that the performance behavior of all processes or threads is largely uniform. Empirical performance-modeling tools such as Extra-P exploit this common pattern to make their modeling process more noise resilient, mitigating the effect of outliers by summarizing performance measurements of individual functions across all processes. While the underlying assumption does not equally hold for all applications, knowing the qualitative differences in how the performance of individual processes changes as execution parameters are varied can reveal important performance bottlenecks such as malicious patterns of load imbalance. A challenge for empirical modeling tools, however, arises from the fact that the behavioral class of a process may depend on the process configuration, letting process ranks migrate between classes as the number of processes grows. In this paper, we introduce a novel approach to the problem of modeling of spatially diverging performance based on a certain type of process clustering. We apply our technique to identify a previously unknown performance bottleneck in the BoSSS fluid-dynamics code. Removing it made the code regions in question running up to 20 times and the application as a whole run up to 4.5 times faster.**

***Index Terms*—Parallel programming, performance modeling, fluid dynamics**

I. Introduction

Whether it is climate change, artificial intelligence, or quantum physics, scientific progress today requires ever increasing computational power. New, powerful supercomputers are being continuously designed to fulfill the practically boundless demand for high performance computing. Given the costs involved and the scarcity of these resources, it is important that scientific applications are efficient and potential bottlenecks are easily identified and eliminated.

A powerful instrument to understand application behavior is performance modeling, an approach that describes the performance of either entire applications or regions of code in terms of a purely analytical expression. A target metric m (e.g., execution time, energy, or number of floating-point operations) is represented as a function $m = f(x_1, ..., x_n)$ of one or more parameters x_i (e.g., number of cores or the size of the input problem), similar to how the complexity of algorithms is defined. Extra-P [1] is an automatic empirical modeling tool capable of leveraging performance measurements to generate performance models, replacing human expertise with a set of experiments where various configuration parameters are varied in order to measure how their change impacts runtime or other metrics of interest.

While Extra-P has proven useful in understanding the behavior of many applications, one of its current limitations is that it can only be applied to programs with spatially uniform performance behavior, that is, whose parallel processes perform largely the same type of operations at the same time. Although this assumption is true for many HPC applications, Extra-P struggles to accurately model unstructured codes or simulations with complex boundary conditions. Furthermore even codes whose behavior should be spatially uniform in theory can exhibit wait states that let the performance of different processes diverge.

In this paper, we expand the performance modeling approach underlying Extra-P, now allowing it to model parallel programs that exhibit multiple behaviors across processes. Using a specialized clustering algorithm originally developed to quantify external interference in parallel applications sharing a network environment, we identify and bundle together

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. The work has been supported in part by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH16008D, and by the German Research Foundation (DFG) through the Program "Performance Engineering for Scientific Software" (BI 714/6-1), and the *ExtraPeak* project, and by the US Department of Energy under Grant No. DE-SC0015524. Calculations were performed on the Lichtenberg high-performance computer of TU Darmstadt. This project is a collaboration in the framework of the Center for Computational Engineering at TU Darmstadt.

different performance regimes, even if their absolute values differ by orders of magnitude and are affected by noise. We discuss different configuration options to allow developers to properly configure their own experiments and therefore obtain the best possible results.

The key contributions of the paper are:

- An empirical performance modeling workflow that leverages clustering to automatically detect and create differentiated models if the application exhibits spatially non-uniform performance behavior
- A case study in which we apply our new approach to BoSSS, a Discontinuous Galerkin (DG) solver, and shows how it helps identify and remove a serious performance bottleneck, resulting in an overall speed up of up to 4.5.

The paper is organized as follows. We first describe Extra-P in Section II and discuss its limitations, before we introduce spatial clustering into its workflow in Section III. We then present BoSSS and the insights we gained after modeling its performance in Section IV and quantify the performance increase achieved by removing the bottlenecks in Section V. Finally, we discuss the broader field of performance modeling in Section VI, and summarize our results in Section VII.

II. Extra-P

Extra-P [1] is a tool capable of creating human-readable performance models out of empirical measurements. These measurements are usually gathered using Score-P [2], an instrumentation framework that gathers performance data at the level of individual function call paths. The core assumption of Extra-P is that the complexity of most function calls can be expressed using the *performance model normal form* (PMNF) 1, that is, effectively with up to n terms composed of a combination of polynomial and logarithmic terms:

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

The PMNF models the effect of parameters x on a response variable of interest $f(x)$. The sets $I, J \subset \mathbb{Q}$ from which the exponents i_k and j_k are chosen and the number of terms n define the discrete model search space.

This expression can be expanded to allow the impact of multiple configuration parameters such as problem size and process count to be modeled at the same time. This is critical, as understanding how such parameters interact is vital in understanding performance bottlenecks. The difference between additive and multiplicative interaction can translate to orders of magnitude more core hours being required to run a program.

$$f(x_1, \ldots, x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \tag{2}$$

At its core, the method uses one data point for each execution configuration as an input, determining coefficients for each potential model using linear regression and cross-validation. Each data point may possibly average several repeated executions to counter noise. In this work, we leverage a recent enhancement of Extra-P, a heuristic [3] that gives users more freedom in the configuration of the measurement space when modeling multiple parameters simultaneously. Previous versions of the tool required all combinations of all values for each parameter to be considered, which, depending on the application, can be difficult. With the help of a method derived using machine learning, models are generated from a smaller subset of measurements, often reducing the cost of measurements by one or two orders of magnitude while at the same time lessening the constraints placed on the experiment design.

```
void compute(float A[Rows][Columns], int myRank) {
    if (position(myRank)==TOP_EDGE)
        computeTopBoundary(A,myRank);

    if (position(myRank)==BOTTOM_EDGE)
        computeBottomBoundary(A,myRank);

    if (position(myRank)==LEFT_EDGE)
        computeLeftBoundary(A,myRank);

    if (position(myRank)==RIGHT_EDGE)
        computeRightBoundary(A,myRank);

    computeInnerValues(A,myRank);
}
```

Listing 1: Simplified code example for a two-dimensional simulation with boundary conditions.

While this particular improvement was necessary in our analysis of BoSSS, over the years many other refinements were made and several applications beyond simple performance modeling were discovered. However, one basic assumption remained: the performance behavior of the target program must be spatially uniform, that is, each process is expected to adhere to the same global performance model. The reason for this constraint is that Extra-P must reduce the measurements to just one data point (for each function call path) in each configuration, while the instrumentation provides one data point for each MPI process or thread. The measurements such as those for runtime taken from individual threads or processes metrics are either averaged or added, depending on the modeling objective—wall time or core hours. Although further statistical tools such as confidence intervals and standard deviation are available, the approach does not support more than one behavior type to be differentiated and modeled, for reasons that will become apparent in the next section. Therefore, if different subsets of processes behave differently, their behaviors are collated, leading to potentially incorrect models.

III. Clustering Empirical Performance Models

In this section, we introduce our clustering approach for empirical performance modeling by first providing a simple example highlighting the challenges of clustering performance

behavior and then showing how our chosen approach can overcome these challenges.

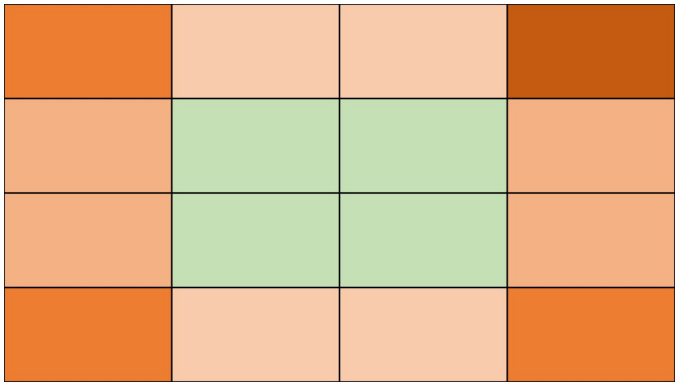
A. Challenges of clustering

The code in Listing 1 shows a simple two-dimensional simulation whose rectangular domain is divided into rectangular blocks, each assigned to a parallel process. For simplicity, we assume that the number of processes is such that the domain can be divided in exactly equal sub-domains across all processes. As will be explained in Section IV, evaluating boundary cells is more expensive in comparison to inner cells. Therefore, we can subdivide the domain in three areas of different cost: all processes will do some baseline computation on their values, however all processes assigned blocks at the edge of the domain must do some additional work for each such edge they contain. This effectively means that processes at the corner will have two such additional computations to perform, the remaining edge processes will have one additional computation to perform while processes assigned inner parts of the domain will only need to perform the baseline computation. By itself, this already defines three different types of behavior.

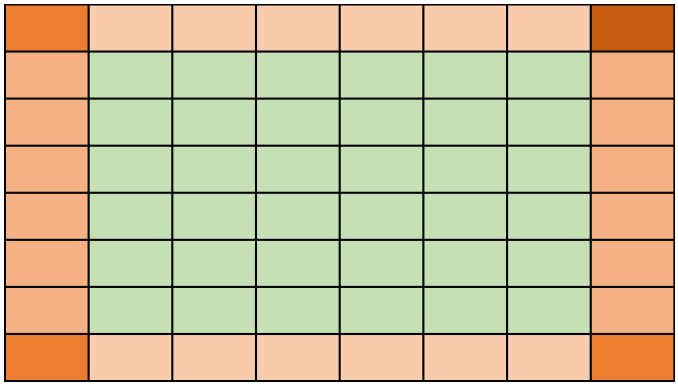
If the area each process covers is rectangular rather than a perfect square, there will be more boundary cells on either the top and bottom edges or the left and right edges, depending on the orientation of the rectangle. This will lead to these types of boundaries to exhibit different behaviors, but can be avoided if the area is a square. However, even in this case, considering a computational domain mapped as a simple two dimensional array, accessing arrays by row or column can have significant differences in performance, forcing us to consider the ”top” and ”bottom” boundary computations as potentially different to the ”left” and ”right” boundary computations, leading to a total of four behavior types. Although this application is not a coupled simulation with different models running on different partitions of the available system, different subsets of the processes clearly show different performance behavior. To understand application performance, this divergence must be taken into account or issues might remain undetected.

The number of processes exhibiting a class of behavior does not necessarily remain constant across different parameter configurations, as shown in Figure 1, further increasing the difficulty of classifying these behaviors unless the rules for classification are already known. In the example, as long as at least four processes are used, there will always be four and only four corner processes regardless of the total number of MPI ranks.

The number of processes with one boundary computation to perform will grow with the square root of the process count for a two-dimensional domain, like in our example, or with the cubic root of the process count for a three-dimensional one. The exact rate depends on the shape of the rectangular domain, but can grow linearly if the domain is effectively one-dimensional rather than two-dimensional. Because the latter is an atypical and rather simple corner case, we choose to focus on a two-dimensional domain in the following analysis.



(a) Topology with 16 processes.



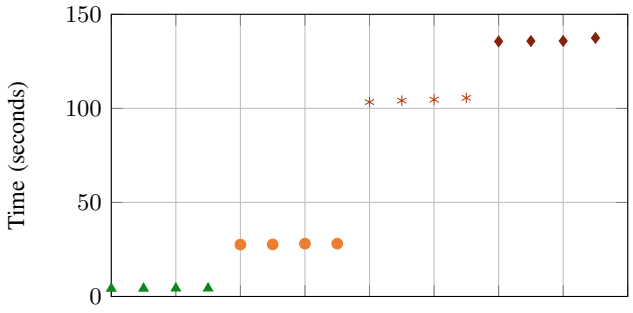
(b) Topology with 64 processes.

Fig. 1: A two-dimensional simulation domain composed of a very high number of individual cells (not pictured), divided among 16 and 64 processes, respectively. Black lines delineate the sub-domain assigned to each process. The color of a sub-domain reflects its computational effort, ranging from green for low to dark orange for high effort.

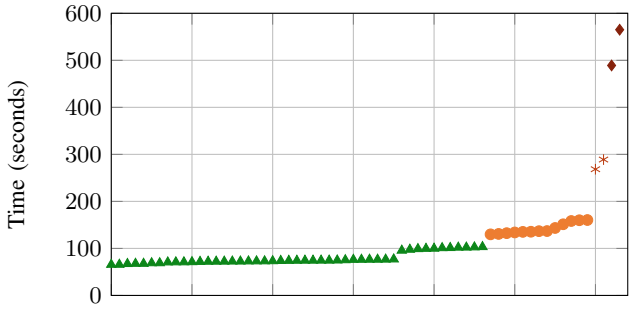
Finally, the inner processes will grow linearly with the number of processes, and will account for the vast majority of behaviors as the program is scaled up. However as such applications usually require communication after the computation step having the vast majority of processes finish faster will not improve performance as the application as a whole will still need to wait for the slower corner processes to finish.

The capability of generating differentiated models representing these behaviors can help developers target their optimization efforts and ensure that performance across the entire application does not suffer because fast processes have to wait for slower ones. Clustering is a natural approach toward differentiating such behaviors. However, developers cannot be expected to know in advance how many behaviors their application exhibits in practice.

Furthermore, the number of behaviors might not even be constant across all parameter configurations. In our example, if we only have 4 processes available, then all will exhibit the same type of behavior, that of a corner process. Therefore, we must automatically decide how many clusters we wish to create only from measurement data.



(a) Per process runtime of the density flux routine in the BoSSS application for 16 processes and problem size 25,600.



(b) Per process runtime of the density flux routine in the BoSSS application for 64 processes and problem size 409,600.

Fig. 2: Comparison of runtime histograms for different routines in the BoSSS application. Data points with the same shape and color are clustered together using our default relative-distance clustering algorithm.

Measurement data is usually affected by noise. The effects of noise on data can be absolute, adding a random overhead to the runtime with a maximum and usually small amplitude, or relative, increasing the runtime by a percentage. Many clustering approaches define an absolute distance to classify data into different clusters. This can be a tremendous difficulty given that the noise affecting larger behavior classes might be more than the difference between smaller behavioral classes. If alongside runtime information hardware counter measurements are available such as the number of instructions or cache misses, these can be used to support the clustering method and improve results.

B. Clustering with relative distance

Figure 2 shows two examples of runtime sets for two different function in the BoSSS application. Clustering would be simple for the first function, as there is little noise affecting the measurements and the differences between clusters is obvious. The second function is more difficult to analyze. The difference between the two largest values is much larger than any difference between the vast majority of values, effectively placing these values in different clusters.

We therefore adopt the clustering approach with relative distance introduced by Shah et al. [4], which takes into account the values of the measurements when considering distances between them and allows accurate clustering even in the presence of noise.

This algorithm performs a single pass on all measurements and only requires that the data clustered to have a total order, which both integers and floating point numbers representing counters and runtime measurements have. It defines the relative distance between two points as their distance divided by the smaller of the distances of the two points from the origin. The algorithm sorts all data and starts with the smallest value in an initial cluster. It then compares the relative distance between each two adjacent points in the sorted list iteratively with a fixed threshold. If the relative distance is below the threshold no new cluster is formed, otherwise a new cluster is formed containing the last point. In either case, the process continues until all points have been assigned to clusters. In our work we set the threshold at 10%. However, too much noise can still be a challenge, which is detailed in Section V.

Summarizing the discussion above, we now explain how the clustering of measurements is integrated into Extra-P's performance modeling process. Figure 3 provides an overview, and below we detail each individual step.

First, the user must select which configuration parameters should be varied, and the values the individual parameters should take. In the overwhelming majority of cases, these parameters will be a measure of the degree of parallelism such as the number of MPI ranks or the number of parallel threads and the size of the problem the program is attempting to solve. In practice, five values for each parameter ensure good model quality. Recent developments of Extra-P [3] allow accurate models to be generated by providing as few as ten measurements when two parameters are considered. Once the users have completed the experiment design, they collect measurements by running an instrumented version of the application in the selected configurations.

Then, the clustering is introduced as an automatic preprocessing step before the actual modeling takes place. If a single cluster is detected, the modeling can proceed as before. If multiple clusters are detected, we provide this information to the user as it is already a valuable insight. It is important to note that we can provide this information without requiring any additional input from the user beyond what the established empirical performance modeling method uses.

We provide additional insights to the user beyond whether clusters exist on a best effort basis: we first analyze if the number of clusters is the same in all measurements. If it is possible to match clusters across all measurement, by default we assume that clusters should be matched in ascending order of the values they contain. If users can supply additional information, we can feed their rules into the matching process, such as changing the threshold used to create new clusters or even place specific processes into clusters. Generating differentiated models for each cluster can help users decide where to direct optimizations efforts, as having parts of the application scaling differently is likely to lead to load imbalance and/or waiting times.

Our tool needs a way to match the clusters across all

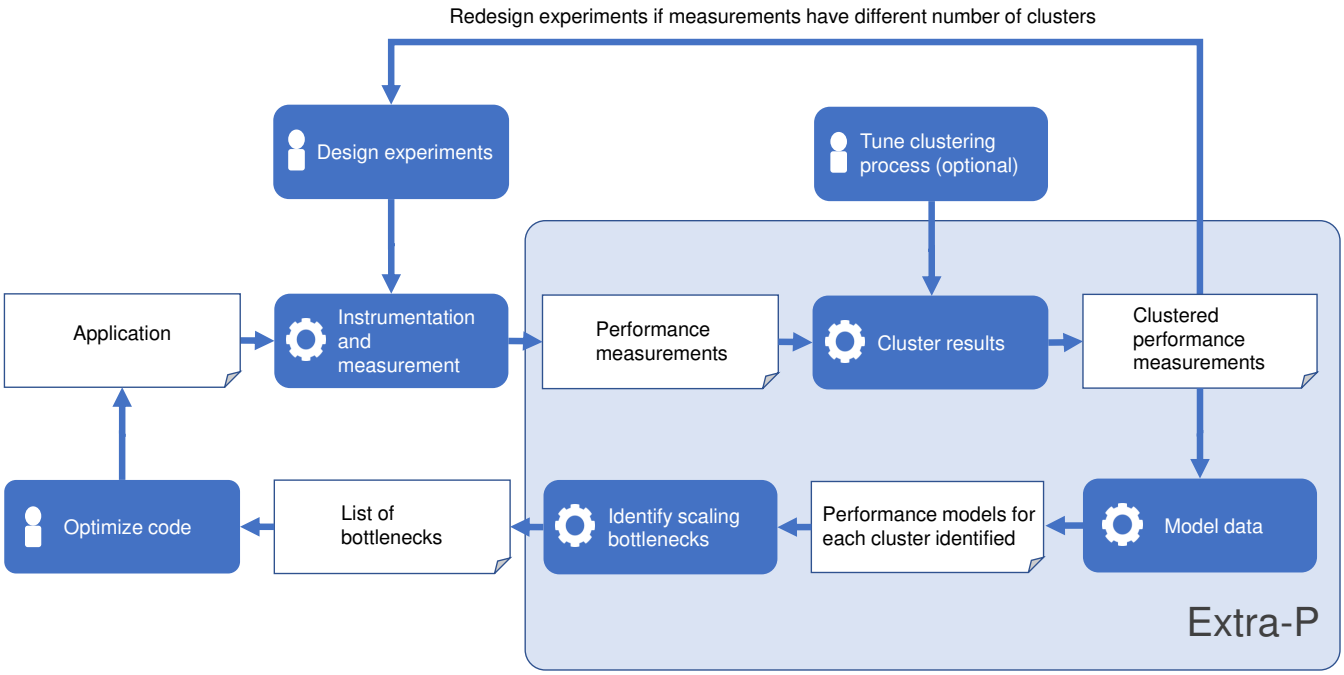


Fig. 3: Workflow for modeling spatially diverging performance. The person symbol in a step indicates that it is manual. A cog wheel represents an automatic step. The addition of the clustering step provides additional value at no extra cost for the user.

measurements to create differentiated models for each class of behavior. If the number of clusters differs across measurements, then this is impossible as we cannot decide which clusters belong together and the users need to redesign their experiments and provide a set of measurements from which the same number of clusters is generated. Alternatively, if sufficient measurements are available where the number of clusters is the same we can create performance models only using that subset. However, even the information of how clusters appear and evolve across measurements can provide many insights to developers, as we will discuss in Section IV.

In conclusion, our clustering can in some cases provide additional information at no additional cost to the user, and can provide even more information if the user provides additional knowledge about what type of behaviors can be found in the code.

IV. BoSSS

We showcase our clustering algorithm by analyzing the performance of a computational fluid-dynamics simulation framework, BoSSS (Bounded Support Spectral Solver). While BoSSS is designed to support arbitrary systems of conservation laws, the present paper is concerned with the Navier-Stokes equations for compressible flows. The employed Discontinuous Galerkin (DG) methods for hyperbolic conservation laws started to gain popularity in the late 1980's and 1990's, through contributions of Cockburn and Shu [5], [6], although its origins can be traced back to the 1970's [7].

A. DG methods for compressible flows

The compressible Navier-Stokes equations can be written in the form

$$\partial_t U + \nabla \cdot F(U, \nabla U) = 0. \tag{3}$$

Here, U denotes the vector of dependent variables, that is, density, momentum, and energy. F is the flux of each of these conserved properties, e.g., the mass flux in the continuity equation.

Like most numerical methods, DG requires a discretization of the flow domain Ω into a numerical mesh with tetrahedrons and/or cubes, labeled herein after as cells $K_0, \dots, K_{J-1}$. In each of the cells, a field property like the density ρ is approximated by a weighted sum over predefined polynomials:

$$\rho(t, \vec{x}) \approx \rho_h(t, \vec{x}) := \sum_{j=0}^{J-1} \sum_{n=0}^{N_k-1} \phi_{jn}(\vec{x}) \tilde{\rho}_{jn}(t) \tag{4}$$

Here, $\phi_{jn}(\vec{x})$ denotes multivariate polynomials, which form a complete basis of the polynomial space of a certain, user-defined degree k. The numbers $\tilde{\rho}_{jn}$ are called the DG coefficients; index j corresponds to a cell, index n is commonly known as a mode index. The support of ϕ_{jn} is limited to cell K_j, that is, $\phi_{jn} = 0$ outside of K_j. Therefore, in the implementation, the sum over j in Eq. 4 can usually

be omitted. Note that the number of polynomials per cell (number N_k) is small, usually below 100, while the number of cells (index j) can be several millions, depending on the discretization and resolution used in the simulation.

The physical setup of the simulation, the so-called *double Mach reflection* is shown in Figure 4. A DG scheme is obtained by inserting the Ansatz (4) into the conservation law (3), applying integration-by-parts and the introduction of so-called numerical flux functions aka. Riemann solvers [8]. To update a specific degree-of-freedom such as $\tilde{\rho}_{jn}$ an explicit scheme of the form

$$\tilde{\rho}_{jn}(t^1) = \tilde{\rho}_{jn}(t^0) - \Delta t \left(\underbrace{- \int_{K_j} F \cdot \nabla \phi_{jn} \mathrm{dV}}_{=:\mathcal{F}_{\text{vol}}} + \underbrace{\oint_{\partial K_j \setminus \partial \Omega} \hat{F}_{\text{int}} \phi_{jn} \mathrm{dS}}_{=:\mathcal{F}_{\text{int}}} + \underbrace{\oint_{\partial K_j \cap \partial \Omega} \hat{F}_{\text{bnd}} \phi_{jn} \mathrm{dS}}_{=:\mathcal{F}_{\text{bnd}}} \right) \quad (5)$$

is used. For the sake of simplicity, we present here only an explicit Euler scheme in order to update from the known state $\tilde{\rho}_{jn}(t^0)$ at time t^0 to the new state $\tilde{\rho}_{jn}(t^1)$ at time t^1. In practice, usually a Runge-Kutta scheme of higher temporal accuracy is used. The integrals on the right-hand-side of Eq. 5 are evaluated numerically using a Gauss quadrature rule of at least order $3k$. The updates can be split up into three parts: First we determine the volume contribution $\mathcal{F}_{\text{vol}}$, which employs the local flux F (cf. Eq. (3)). It only depends on data associated with cell K_j and produces a uniform computational load across all cells. Second we determine the flux on interior cell boundaries $\mathcal{F}_{\text{int}}$ and third we determine the flux on domain boundaries $\mathcal{F}_{\text{bnd}}$, which depend on the respective Riemann solvers $\hat{F}_{\text{int}}$ and $\hat{F}_{\text{bnd}}$. $\hat{F}_{\text{int}}$ ensures numerical coupling of the individual cells and therefore depends on data associated with *two* adjacent cells. $\hat{F}_{\text{bnd}}$ incorporates the boundary condition: to ensure numerical stability first a 'virtual' exterior state is computed and then the flux at the boundary via $\hat{F}_{\text{bnd}}$ [9]. Therefore, the computation of $\hat{F}_{\text{bnd}}$ is significantly more expensive than $\hat{F}_{\text{int}}$.

Considering again Figure 1, it becomes obvious that this causes in-homogeneous computational load per cell: For interior cells, where the intersection of cell and domain boundary is empty, that is, $\partial K_j \cap \partial \Omega = \emptyset$, the boundary flux $\mathcal{F}_{\text{bnd}}$ has no impact. For cells located at the edges, resp. the corners of the domain Ω, the more expensive boundary Riemann solver $\hat{F}_{\text{bnd}}$ is invoked at 25 % resp. 50 % of the quadrature nodes at the cell boundary.

B. C# for High Performance Computing

C# was chosen as the main language for developing BoSSS, as C# is almost as versatile and simple as Python, but can deliver, if used carefully, performance closer to C. Several of the compute-intensive functions such as the evaluation of the flux F as well as the Riemann solvers $\hat{F}_{\text{int}}$ and $\hat{F}_{\text{bnd}}$ are implemented in C#. Optimized BLAS subroutines are used to accelerate certain computations: For example, the sum in Eq. (4) has to be evaluated in all cells, for the same sets of nodes in each cell. This can be rewritten as a matrix-matrix operation, enabling the use of the general dense matrix-matrix multiply (*DGEMM*).

The execution environment of C# provides many of the commonly used functionalities, including data structures, file system, and networking. Also, the virtual machine provides a just-in-time compiler to generate efficient code when executing applications written in C#. Moreover, the language provides a layer to link to and call native libraries.

Unlike languages that are compiled to native code immediately, C# is first translated into an abstract machine model, the so-called byte code, sometimes also referred to as bit code. This abstract machine model provides binary platform independence with a minimum of platform-specific execution branches, and allows the transfer of executables between different operating systems and even hardware architectures. At runtime, the byte code is executed by a virtual machine.

To enable multi-million cell simulations, BoSSS is fully MPI-parallelized, exploiting the aggregate memory of many dedicated compute nodes. The required MPI routines, typically highly-optimized native-code libraries, are linked to C# via P/Invoke directives. Binary compatibility with both major MPI families, MPICH (with its variants Microsoft MPI and MVAPICH2) and Open MPI, is achieved using the Fortran bindings of MPI. This is necessary because certain implementation details are handled differently in the C interface. For example, in MPICH communicators are implemented as an integer constant, whereas in Open MPI it is a pointer. But both Fortran interfaces handle them in a similar fashion (both use integer values).

V. Performance Evaluation

BoSSS has over 1.07 million lines of code and over 17.1 thousand functions. In this work, we focus on the compressible Navier-Stokes solver which is used to simulate the Double-Mach reflection, contains over 16 thousand lines of code and over 100 methods itself but calls methods from other parts of the framework as well. We chose this solver as the developers use it extensively and had clear performance expectations for it. To model the performance of BoSSS empirically, we ran it with all combinations of MPI processes in the set $p \in \{4; 8; 16; 32; 64; 128; 256\}$ and problem size per core $s \in \{1,600; 6,400; 25,600; 102,400; 409,600\}$, repeating each configuration four times to counter noise. The measurements were gathered on the Lichtenberg Cluster in Darmstadt. The cluster contains two types of nodes: nodes containing 8 sockets with 2 Intel Xeon E5-2670 AVX processor cores each with either 32 or 64 GB memory per node and nodes containing 12 sockets with 2 Intel Xeon E5-2680 v3 AVX2 processor cores each with 64 GB memory per node.

A preliminary analysis revealed that four functions that were repeatedly called by the application had both

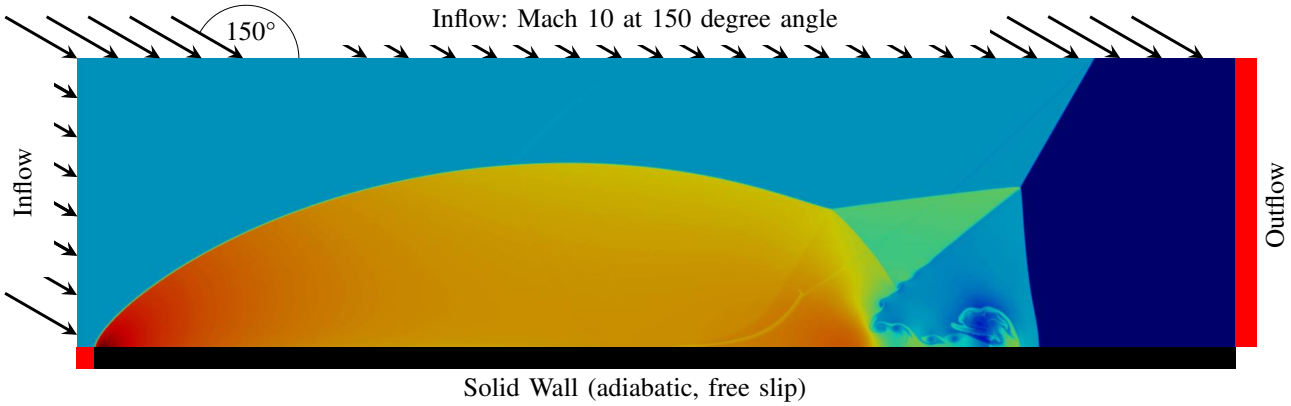


Fig. 4: *Double-Mach Reflection*: A sloped shock impacts on a solid wall, this is an often employed benchmark for compressible flow solvers. The setup is the equivalent of a vertical shock wave at Mach number 10 impinging on a ramp that is inclined at an angle of 30 degree. As the shock impacts the wall, it is reflected in a complex fashion. Self-similar flow pattern are developing, featuring different types of shock-wave interactions.

TABLE I: Performance models for each of the behavior classes in functions of BoSSS identified as performance bottlenecks prior to optimization. We include the median runtime at $p = 64; s = 409,600$ to provide a intuitive picture of the difference between cluster runtimes. $\hat{R}^2$ is the adjusted coefficient of determination.

Kernel and cluster type	Model with default clusters	$\hat{R}^2$	Model with tuned clusters	$\hat{R}^2$	Median at runtime
Density flux					
Corner	$18 \cdot \log s - 2 \cdot \log p$	0.77	$0.17 \cdot \log s \cdot \log p + 16 \cdot \log p$	0.65	127s
Top& bottom edge	$4 \cdot 10^{-3} \cdot s^{0.8} - 2 \cdot 10^{-4} \cdot s^{0.8} \cdot \log p$	0.72	$2.1 \cdot \log s \cdot \log p - 24.6 \cdot \log p$	0.58	109s
Left& right edge	$6 \cdot 10^{-2} \cdot s^{0.33} \cdot \log s$	0.90	$2.7 \cdot 10^{-3} \cdot s^{0.67} \cdot \log p - 2.1 \cdot \log p$	0.87	85s
Inner	$1.5 \cdot 10^{-4} \cdot s$	1	$1.5 \cdot 10^{-4} \cdot s$	1	62s
Energy flux					
Corner	$1.3 \cdot s^{0.25} \cdot \log p - 16 \cdot \log p$	0.66	$16 \cdot \log s$	0.60	141s
Top& bottom edge	$1.9 \cdot 10^{-7} \cdot s^{1.33} \cdot \log s + 0.01 \cdot \log p$	0.77	$11.5 \cdot \log s$	0.52	127s
Left& right edge	$1.2 \cdot 10^{-3} \cdot s^{0.75} \cdot \log p - 1.1 \cdot \log p$	0.91	$2.9 \cdot 10^{-4} \cdot s^{0.75} \cdot \log s$	0.92	102s
Inner	$2.2 \cdot 10^{-4} \cdot s$	1	$2.2 \cdot 10^{-4} \cdot s$	1	88s
Momentum flux					
Corner	$2.8 \cdot \cdot s^{0.25} \cdot \log p - 35 \cdot \log p$	0.67	$34.9 \cdot \log s$	0.65	305s
Top& bottom edge	$7.6 \cdot 10^{-6} \cdot s^{1.33}$	0.68	$25.2 \cdot \log s$	0.56	276s
Left& right edge	$4 \cdot 10^{-3} \cdot s^{0.67}$	0.91	$6.3 \cdot 10^{-3} \cdot s^{0.8}$	0.93	224s
Inner	$4.8 \cdot 10^{-4} \cdot s$	1	$4.8 \cdot 10^{-4} \cdot s$	1	195s
Viscosity flux					
Corner	$7.3 \cdot 10^{-4} \cdot s + 8.8 \cdot 10^{-6} \cdot s \cdot \log p$	0.99	$7.7 \cdot 10^{-4} \cdot s$	1	318s
Top& bottom edge	$7.5 \cdot 10^{-4} \cdot s - 2 \cdot 10^{-8} \cdot p^3 \cdot \log p$	0.99	$7.6 \cdot 10^{-4} \cdot s$	1	315s
Left& right edge	$7.5 \cdot 10^{-4} \cdot s - 0.1 \cdot p^{0.25} \cdot \log p$	0.99	$7.3 \cdot 10^{-4} \cdot s$	1	301s
Inner	$7.1 \cdot 10^{-4} \cdot s$	1	$7.1 \cdot 10^{-4} \cdot s$	1	292s

significantly higher runtimes than the developers expected, exhibiting varying numbers of clusters depending on the execution configuration, namely the number of clusters grew as the process count increased. These functions, `density_flux`, `energy_flux`, `momentum_flux`, and `viscosity_flux` are the Riemann solvers $\hat{F}_{\text{int}}$ and $\hat{F}_{\text{bnd}}$ (c.f. Section IV, Eq. 5) for the respective transport equations for mass, energy, and momentum (including viscous effects).

Our first set of experiments only contained combinations of the MPI processes in the set $p \in \{4; 8; 16; 32; 64; \}$ and problem sizes per core $s \in \{1,600; 6,400; 25,600; 102,400; 409,600\}$. While these were sufficient to discover that multiple behaviors are present for numbers of processes greater than four, they did not allow us to model all behavior classes. With four

processes, we only had examples of corner processes, and with 8 processes only corner and top and bottom edges. With sixteen processes or more, all classes of behavior are present. Extra-P requires at least four, but preferably five data points in each parameter dimension to allow accurate models to be derived.

We therefore expanded our measurement set to include MPI processes in the set $p \in \{128; 256\}$. In Table I, we show the performance models of the different behavior classes identified by our clustering approach, using the full set of measurements available.

Unfortunately, clusters with processes working on boundaries are less numerous than inner processes, and their measurements are also more strongly affected by variance. This makes both clustering and modeling these behaviors quite difficult. Figure 5 shows a set of measurements where no definite way of generating clusters is apparent. The small values are very similar and should therefore be clustered together, but the other values grow with jumps and variations making a clear separation into clusters impossible. This negatively impacts the modeling process. While the model for the inner processes is always identified according to developer expectation, models for the other processes vary significantly.

With the developers identifying the issue, namely that processes involved in boundary computations are the performance bottlenecks, we tune the clustering approach to match their expectations. The tuned approach uses rules to assign processes to clusters according to the problem decomposition and number of boundary computations and is more accurate than the default one which only relies on relative distance. In the case of the viscosity flow function this tuned approach generates models in agreement with developer expectations.

To decide how well our models fit the data, we use $\hat{R}^2$, the adjusted coefficient of determination, as shown in Table I. It is an indicator of how much better a given model fits the set of measurements used to derive it, compared to simply averaging all measurements and using the average value instead of the model. It also applies a penalty to models depending on the degrees of freedom they have to limit over-fitting. In our experience, values smaller than 0.95 indicate that the model is not able to fit the data well, and a value of 1 indicates a perfect fit.

Having clusters containing varying numbers of data points affected differently by noise is likely to be an issue for other applications as well. The default clustering algorithm we propose allows good models to be generated if enough data with little noise is available, but should be tuned as the analysis uncovers additional information about how the clustering should be performed. If they are available, hardware counter information such as numbers of instructions and cache misses can further guide the clustering process and allow users to improve the clustering results. This in turn will improve the quality of the performance models we generate from these clusters and hopefully allow further insights to be gained.

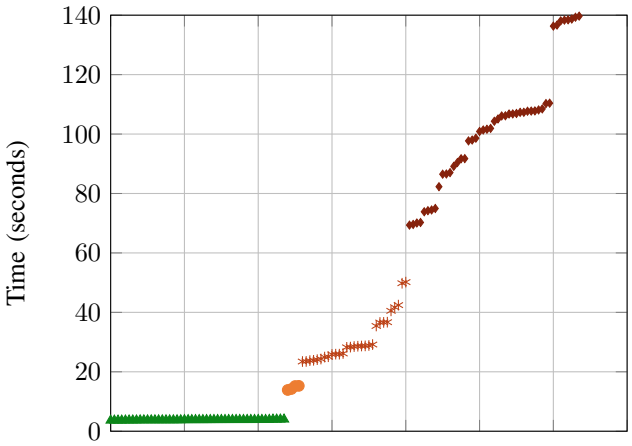


Fig. 5: Example of measurement set that is challenging to cluster. Measurements represent the runtime of the density flow routine across all experiments where $p = 32$ and $s = 25,600$. The colors represent the different clusters generated using the clustering algorithm using relative distance.

A. *Optimization of boundary fluxes*

To explain the performance bottleneck discovered and how it was removed, we now return to the BoSSS code and the physical processes it simulates. In a regular scenario, that is one with between 10^4 and 10^6 cells per core, the interior Riemann solver $\hat{F}_{\text{int}}$ is invoked far more often than $\hat{F}_{\text{bnd}}$ and thus its impact on application runtime was expected to be dominant. Therefore the optimization of the implementation of $\hat{F}_{\text{int}}$ was prioritized.

The divergence in performance models for the four flux computation functions across clusters leads to a more detailed analysis of these functions and their models. We observe that models for processes that must compute boundaries scale worse than inner processes. Each process simulates as part of its sub-domain a very large number of cells. The developers discovered that for these routines, the evaluation of $\hat{F}_{\text{bnd}}$, for a single cell edge is about 80 times more expensive than the evaluation of $\hat{F}_{\text{int}}$. This means that each cell at an egde of the simulation domain takes 80 times longer to compute compared to the inner cells, and the corner cells take up 160 times longer. As the ratios of inner cells to edge cells changes with the number of cells in the subdomain, the runtime will also change resulting in the performance models we observe.

Two reasons were identified that lead to this performance gap: First, the implementation of $\hat{F}_{\text{int}}$ is 'vectorized', that is, it computes the Riemann solution for multiple quadrature nodes in a single function call, which hides the overhead of calling virtual functions. The implementation of $\hat{F}_{\text{bnd}}$ did not have such an optimization. Second, the implementation of the computation of the virtual exterior state is rather complex—it involves several evaluations of trigonometric functions as well as square roots.

To close the performance gap between the evaluation of the interior and the boundary edge flux, the implementation

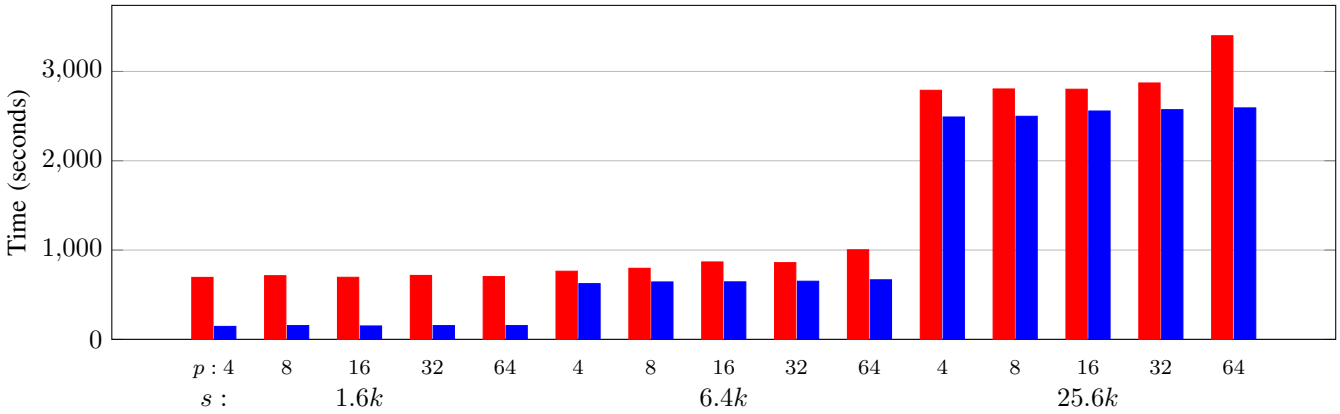


Fig. 6: Total application runtime before (left bar) and after optimizing the performance bottleneck identified through performance modeling (right bar).

TABLE II: Performance models for functions identified as performance bottlenecks in BoSSS after optimization. Note that the behavior can no longer be separated into different classes.

Kernel	Model	$\hat{R}^2$
Density flux — All processes	$1.7 \cdot 10^{-4} \cdot s$	1
Energy flux — All processes	$2.3 \cdot 10^{-4} \cdot s$	1
Momentum flux — All processes	$5.2 \cdot 10^{-4} \cdot s$	1
Viscosity flux — All processes	$7.3 \cdot 10^{-4} \cdot s$	1

of $\hat{F}_{\text{bnd}}$ was vectorized in the same fashion as $\hat{F}_{\text{int}}$. This also allowed a couple of optimizations in the computation of the virtual exterior state. For example, we are now able to cache certain results of trigonometric functions. After these optimizations, the evaluation of $\hat{F}_{\text{bnd}}$ is 'only' three times more expensive than $\hat{F}_{\text{int}}$. For the investigated number of MPI processes, these modification have shown to be sufficient in order to ensure that the evaluation of boundary conditions no longer dominates the application runtime.

B. Performance improvement

After optimizing the four bottleneck functions `density_flux`, `energy_flux`, `momentum_flux`, and `viscosity_flux`, as discussed in the previous subsection, we can observe the significantly improved performance models for these functions in Table II. The boundary computations have been optimized to the degree that measurements no longer detect any meaningful difference between the computations performed by the inner processes compared to processes with one or more boundaries to simulate.

Furthermore the performance models we identify are, without exception, of the same complexity class as the models for processes without any boundary condition to compute (i.e., inner processes) from the unoptimized version of the code. Therefore the optimized boundary computation has no measurable impact on the overall performance of BoSSS. Figure 6 shows a comparison of the optimized and unoptimized version of BoSSS for different configurations of process count and problem size. For $s = 1.6k$ the improvement is 4.5 times larger, and on the largest problem and process count we have measurements available for it is an improvement of 30%. We therefore confidently claim that our performance analysis workflow has allowed the developers to pinpoint and eliminate a significant bottleneck in their application and substantially improve the runtime of BoSSS.

VI. Related Work

Analytical modeling is an approach with a long history of successes in performance analysis of high performance applications [10], [11]. While requiring significant effort on the part of the human experts significant insights into complex behaviors can be gained through manual code analysis [12]–[14]. Of particular relevance to our work is the discovery that large differences between actual and predicted performance can be caused by system noise (Petrini et al. [15]). We use the six-step process to create application performance models defined by Hoefler et al. [16] as a basis for our own, automatic, approach.

Several methods automate parts of the performance modeling process in the attempt to make it easier to use on real applications [17]–[20]. Siegmund et al. [21] also use methods derived from statistics, however, by focusing on applications monolithic-ally they cannot pin-point bottlenecks or differentiate behaviors in the way our approach does.

Clustering has long been used as a tool in the performance analysis arsenal. A large survey by Isaacs et al [22] provide an overview of different performance visualization techniques and how they categorize performance information. Gamblin et al [23] propose CAPEK, a parallel clustering algorithm that enables in-situ analysis of performance data at run time. Ahn et al. [24] suggested the use of clustering as a way to understand multi-variate relationships in measuring hardware counters. Huck et al. [25], [26] use clustering to determine representative behaviors automatically out of large performance experiments

and provide a tool, PerfExplorer to achieve this goal. Our own clustering approach is based on recent work by Shah et al. [4], that introduces a relative distance metric rather than absolute thresholds to allow the correct clustering of data sets affected by noise with large differences between clusters. This advance was necessary to estimate application interference in shared network environment but can be adapted for empirical performance modeling as well.

VII. Conclusion

After introducing spatial clustering in our empirical performance-modeling workflow, we can now create differentiated models, representing the distinct behavior classes being observable on different MPI processes during the same execution of a parallel application. This feature supports the discovery of spatial performance divergence related properties of the process topology. Its application was essential in uncovering a previously unknown performance bottleneck in the BoSSS fluid dynamics code. Its removal left the affected function 20 times faster than before, resulting in an overall speed-up for the entire application of up to a factor of 4.5. We plan to integrate this feature into the next release of Extra-P, enabling even more developers to uncover performance bottlenecks related to spatial divergence.

References

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

[2] 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*, Gauß-Allianz. Springer, 2012, pp. 85–97.

[3] M. Ritter, A. Calotoiu, S. Rinke, T. Reimann, T. Hoefler, and F. Wolf, "Learning cost-effective sampling strategies for empirical performance modeling," in *Proc. of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA*. IEEE Computer Society, 2020, (to appear).

[4] A. Shah, M. Müller, and F. Wolf, "Estimating the impact of external interference on application performance," in *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, ser. Lecture Notes in Computer Science, M. Aldinucci, L. Padovani, and M. Torquati, Eds., vol. 11014. Springer, 2018, pp. 46–58. [Online]. Available: https://doi.org/10.1007/978-3-319-96983-1_4

[5] B. Cockburn and C.-W. Shu, "TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws II: General Framework," *Mathematics of Computation*, vol. 52, no. 186, p. 411, Apr. 1989. [Online]. Available: http://www.jstor.org/stable/2008474?origin=crossref

[6] ——, "The P1-RKDG Method for Two-dimensional Euler Equations of Gas Dynamics," ser. ICASE Report 91-32, 1991.

[7] W. H. Reed and T. R. Hill, "Triangular mesh methods for the neutron transport equation," in *National topical meeting on mathematical models and computational techniques for analysis of nuclear systems*, ser. CONF-730414–2; LA-UR–73-479. Los Alamos Scientific Lab., N.Mex. (USA), 1973, p. 23.

[8] D. A. Di Pietro and A. Ern, *Mathematical Aspects of Discontinuous Galerkin Methods*, ser. Mathématiques et Applications. Springer, 2011, no. 69.

[9] J. van der Vegt and H. van der Ven, "Slip flow boundary conditions in discontinuous Galerkin discretizations of the Euler equations of gas dynamics," National Aerospace Laboratory NLR, Tech. Rep. NLR-TP-2002-300, 2002.

[10] 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)*. ACM, 2001, p. 37.

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

[12] 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)*, 2007, pp. 279–284.

[13] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 336–346, February 2006.

[14] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," *SIGMETRICS Performance Eval. Review*, vol. 32, no. 1, pp. 2–13, June 2004.

[15] 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*, ser. (SC '03). ACM, 2003, p. 55. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050204

[16] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*, ser. SC '11. ACM, 2011, pp. 6:1–6:12. [Online]. Available: http://doi.acm.org/10.1145/2063348.2063356

[17] N. R. Tallent and A. Hoisie, "Palm: Easing the burden of analytical performance modeling," in *Proc. of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 221–230. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597683

[18] 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*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 84:1–84:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389110

[19] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical Models for Empirical Search-Based Performance Tuning," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 65–94, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004041293

[20] 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)*, May 2015, pp. 1161–1170.

[21] 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*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 284–294.

[22] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the Art of Performance Visualization," in *EuroVis - STARs*, R. Borgo, R. Maciejewski, and I. Viola, Eds. The Eurographics Association, 2014.

[23] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed, "Clustering performance data efficiently at massive scales," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 243–252.

[24] D. H. Ahn and J. S. Vetter, "Scalable analysis techniques for microprocessor performance counter metrics," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002, pp. 3–3.

[25] K. A. Huck and A. D. Malony, "Perfexplorer: A performance data mining framework for large-scale parallel computing," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 41–41.

[26] K. A. Huck, A. D. Malony, R. Bell, and A. Morris, "Design and implementation of a parallel performance data management framework," in *2005 International Conference on Parallel Processing (ICPP'05)*, 2005, pp. 473–482.