





# Differential Performance Analysis Workflow for Algorithmic Changes

Isabel Thärigen   
IT Center  
RWTH Aachen University  
Aachen, Germany

Joachim Protze   
IT Center  
RWTH Aachen University  
Aachen, Germany

Fabian Orland   
IT Center  
RWTH Aachen University  
Aachen, Germany

Marc-André Hermanns   
IT Center  
RWTH Aachen University  
Aachen, Germany

**Abstract**—Most performance analysis tools used in HPC focus on the analysis of a single configuration of an application. In this work, we instead present a novel performance analysis workflow, supporting the comparison of varied code versions and running conditions. There exist different code versions for many applications because they comprise parts that can be implemented in various ways or already exist in third-party libraries, like linear solvers. Additionally, varied running conditions like scaling of execution units or exchanging the input data can influence the performance behavior. Performance comparison of different application configurations helps determine the best configuration and understand differences in behavior. Such measurements are often not supported directly and are cumbersome to handle manually with current performance measurement and analysis tools. This work presents a workflow based on the Jülich Benchmarking Environment (JUBE) that automatically handles the multitude of measurements and data collation after an initial manual configuration. Furthermore, we introduce diagrams suited for a clear and precise presentation of the collected performance data. The proposed workflow is showcased using two applications *CalculiX* and *Jukkr*. Our application studies highlight that our workflow allows a detailed performance analysis while still being easy to use. We, therefore, encourage integrating our approach of multi-configuration diagrams into broadly used HPC visual performance exploration tools.

**Index Terms**—OpenMP, Performance, Tools, Workflow

## I. INTRODUCTION

Performance measurement and analysis is an essential part of parallel computing, as it allows to determine what influences an application’s runtime or resource consumption the most. There exists a multitude of different tools specialized in performance measurements and analysis. While they offer great possibilities to measure or analyze a single version of an application, it is hard to compare multiple versions with each other. Tasks like matrix multiplications or linear equation solving are often outsourced to third-party libraries. This leaves a developer with the challenge to determine which third-party implementation offers the best performance when integrated into the own application.

Algorithmic changes, such as swapping third-party libraries in an implementation, result in different variants of the same application, but leave the general structure intact. Therefore, a comparison between runs of different versions of an application is feasible and valuable. Apart from integrating third-party code, we can also understand the analysis functionality

introduced by runtime analysis tools like tracing tools or correctness tools as algorithmic changes.

In many cases, proxy apps are used to study performance behavior and tune the performance of individual parts of an application. When deriving the proxy app from the original code and when porting back the tuned version to the original code, it is crucial to verify that the proxy app reflects the application’s behavior. The kernel of interest in the proxy app and the original code should not differ algorithmically. Therefore, we expect the same performance behavior for both code versions, which we can also verify with our workflow.

To the best of our knowledge, there exists no tool so far that supports such a comparison. The contributions of this work comprise (a) a novel workflow using JUBE [1] to execute external performance tools and store information from multiple runs easily, (b) proposals for the analysis of the collated information, and (c) a diagram-based approach that clearly and concisely depicts the relevant information. The framework is available on GitHub <sup>1</sup>.

The rest of this paper is structured as follows: Section II gives a short introduction to JUBE and hardware performance counters, as used in the workflow. The following Section III provides an overview of existing performance measurement and analysis tools and discusses why they do not provide the desired functionalities. In Section IV an overview of our novel workflow is given. The workflow can be divided into two steps data collection and data processing, which are explained in more detail in Sections V and VI respectively. The complete workflow is showcased using mini-apps from two selected parallel applications *CalculiX* and *JuKKR-KKRhost* in Chapter VII. Finally, we conclude our work in Chapter VIII.

## II. BACKGROUND

### A. The Jülich Benchmarking Environment

To drive our workflow, we use the Jülich Benchmark Environment (JUBE) [1]. JUBE is a Python application developed at Forschungszentrum Jülich, that manages a hierarchical workflow based on XML configuration files. As the name suggests, its origins lie in benchmarking workflows evaluating the performance of a given set of applications on an HPC system. It also lends itself to use its extended capabilities

<sup>1</sup><https://github.com/RWTH-HPC/diff-perf-framework>

for driving an in-depth performance analysis workflow of an application or runtime system. At its core, JUBE manages the execution of one or more steps with potential parent-child dependencies, quickly enabling the evaluation of parameter spaces by branching instances of a step with the cross-product of all multi-value parameters.

**Listing 1** Two multi-value parameters in a JUBE parameter set. The value separator defaults to comma, but can be set to a different character.

```
<parameterset name="example_pset">
  <parameter name="param1">1,2</parameter>
  <parameter name="param2">2,4</parameter>
</parameterset>
```

Listing 1 shows a JUBE *parameter set* containing two parameters `param1` and `param2` with each containing a comma-separated list of values. JUBE will generate the cross-product of instances with a specific single value realization for each multi-value parameter combination in the step where this parameter set is initially used. This means for the given example that JUBE would create step instances with the (`param1`, `param2`) combinations (1,2), (1,4), (2,2), and (2,4). Instances of dependent (child) steps will inherit each parameter's specific realization in the parent step. Therefore, depending on the step a parameter set is first used in, one can create a tree of different branching levels, covering the parameter space that is supposed to be explored at that workflow level.

For each instance of a step, JUBE creates a separate sandbox directory such that each instance has a clean working directory. Each step instance has easy access to output generated in a step instance of its specific ancestry using filesystem links to the parent sandbox. Furthermore, JUBE also allows creating shared steps performed once for all siblings of a specific step in the workflow.

Next to the generation and execution of steps, JUBE also provides an easy-to-use infrastructure to gather output data in any generated output of individual steps. Using the gathered data and the specific realization of all parameters for a given path in the tree of steps, JUBE can generate output in either CSV format or a more human-accessible pretty printed ASCII table.

### B. Hardware Performance Counters

Most modern processors have a small set of counters that can keep track of hardware performance events. There are many different metrics to monitor, but only a handful can be measured simultaneously due to the limited number of counters. The computing system used in the evaluation, for instance, allows up to eight hardware counters at a time. The metrics generally count performance-related hardware events like memory accesses or specific instructions.

Hardware performance counters offer a great possibility to determine reasons for performance differences. In our workflow, we let the user choose the hardware performance counters they want to track and define a specific set of hardware performance counters to characterize the cache usage.

Different high-level APIs such as LIKWID [2] and PAPI (Performance API) [3] support measuring hardware performance counters by abstracting away the platform-specific details and differences. Alternatively, a tool can also directly use the `perf_event` kernel API. While we could use any of these hardware performance counter APIs in our workflow we use PAPI in our prototype.

### III. RELATED WORK

Performance analysis is an important topic for modern HPC applications, so there are many different tools to gather and analyze performance data. Score-P [4] is a measurement infrastructure combining the functionalities of already established measurement tools, Scalasca [5], TAU [6] and VampirTrace [7]. Score-P allows to profile or trace an application, either by sampling or using automatic or manual instrumentation. The analysis can be done live or post-mortem, and it supports different types of parallelization like MPI, OpenMP, or CUDA. For post-mortem analysis, Score-P stores the measurement results either in CUBE4 (profile), TAU (profile), or OTF2 (trace) format. CUBE, Vampir, or Extra-P use the collected data for their analysis and presentation.

JUBE [8] is an examination and analysis tool for profile files. Its user interface differentiates between three dimensions: the metric dimension, the system dimension, and the program dimension. Each dimension is displayed in a different segment, which allows it to easily switch between different metrics and system components and observe the performance of different call paths. While command-line tools to manipulate multiple CUBE files using the so-called *CUBE algebra* [9] exist, their output is always a single resulting CUBE report, which can then be explored in the CUBE GUI. However, the CUBE GUI does not support comparing multiple profiles visually except by opening every profile in separate windows and comparing the values by hand. The CUBE GUI does support viewing the difference of two different cube reports (using the CUBE algebra), yet it does not support visualization of performance data beyond pairwise comparison. CUBE therefore does not offer the functionality needed for our desired comparative performance analysis.

Analyzing traces can be done using Vampir [7], which even provides the possibility to open multiple trace files at once in a comparison view. Unfortunately, we realized that opening more than six or sometimes even more than just three traces at the same time leads to illegible results. Additionally, traces provide the complete call history. It would be necessary to align the corresponding regions in order to compare different variants of an application. For example, the second parallel region in the first variant should be displayed at a similar position as the second parallel region in the second variant to compare their metric values. Experiments with an alignment tool presented by Weber [10] did not provide the desired results, so using Vampir and, in general, traces were relinquished.

Another approach is needed to avoid the manual comparison of metric values. Extra-P [11] is a tool designed to generate

scaling models of applications based on measurement values. Similarly, HPCToolkit [12] allows to compare two call-path profiles and extrapolate the scalability. Timemory [13] automates roofline model calculation. All approaches leverage the number of individual values that have to be compared or post-processed but still focus on analyzing one application in one fixed variant. However, to analyze an application's behavior when replacing central algorithms, we need a clear and understandable way to compare the different runtime characteristics.

Hatchet [14] is a data processing tool to process profiling data programmatically. The Python-based library builds on the pandas data analysis library. The library allows to filter, aggregate, and prune datasets collected from parallel application runs.

We propose a novel workflow that automatically measures and collects all data and generates suitable diagrams to compare different variants of an application. Hatchet is closely related to our data processing step, so we might extend Hatchet in the future to integrate differential operations on profiling data and create the diagrams presented in this paper.

#### IV. WORKFLOW OVERVIEW

In this work, we compare different implementation variants of an application. For each variant, the main application stays the same, but some inner algorithms are exchanged. The goal is to compare the different variants against each other in terms of their performance for different thread or process numbers. Figure 1 shows an overview of the complete workflow.

The first necessary step for the analysis is to gather performance data from the application. We measure the runtime and, if desired, hardware performance counters for different thread or process numbers for each variant. In order to deal with the rather large parameter space of the measurements, we propose a novel workflow using JUBE. JUBE takes care of the measurement execution and organization, while gathering the data is done using a profiler. For each new application, the JUBE workflow needs to be configured once at the beginning to specify how the application can be built and executed. A different configuration has to be adjusted for each new system to explain which modules should be loaded and which performance counters should be tracked. The application and system configurations are independent of each other, so if one application is measured on different systems, the application configuration only has to be adjusted once. After the configurations for the application and system are done, all necessary measurements can be executed at once with a single `jube run` command. Repeating the measurements also only takes a single `jube run` command. JUBE stores all measurement results in a run directory, with one folder per application and thread/process number. A more detailed explanation of the JUBE workflow and the performance value gathering will be given in Section V.

The second step of our workflow is data processing, which is needed in order to deal with the large number of measurement values. We chose a diagram-based approach to present the

gathered data. In our prototype implementation, a python script gathers all the data from the JUBE run, generates the diagrams, and stores them in four different PDF files. We see these PDF files as a mockup for possible displays in interactive performance analysis tools. Section VI explains the four different generated types of diagrams.

#### V. DATA COLLECTION

The previous section already gave a brief overview of the complete workflow. This section describes one possibility to gather meaningful data efficiently using a profiler and discusses the novel workflow using JUBE in more detail than before.

##### A. Gathering Data With A Profiler

In performance analysis, we distinguish profiling and tracing data of applications. A trace contains a time series of events, which allows understanding a program's execution's particular time characteristics. Aligning and comparison of traces has been studied before [10], [15]. The comparison of traces is usually limited to a smaller number of traces and not applicable to scaling analysis of two or more program variants, as already explained above. A profile contains aggregated timing information, which allows for identifying hotspot regions, yet lacks information on the evolution of the program behavior. In this work, we focus on the profiling of regions within an application. The data collection tools typically distinguish three kinds of instrumentation to identify regions in a program's execution: (a) Compiler instrumentation, (b) manual user instrumentation, and (c) runtime instrumentation. Compiler instrumentation marks the entry and exit of functions so that the instrumented functions appear as a region. User instrumentation is an interface, which allows the application programmer to define and mark regions manually. Runtime instrumentation uses tool targeted interfaces of (parallel) runtime libraries such as PMPI for MPI or OMPT for OpenMP to mark regions.

In general, any instrumenting profiler can be used in our proposed workflow, and we need an adapter to collect and feed the data into our evaluation step. Examples for commonly used profilers are Score-P [4], Extrae [16] or TAU. HPCToolkit [12] also supports profiling but usually collects the data sample-based rather than instrumentation-based, which can lead to wrong conclusions in our comparative analysis.

For our prototype, we decided to implement a simple profiler inside LLVM using the OMPT interface [17]. This profiler allows us to analyze the performance impact of the above profiling tools and analyze the latest OpenMP features, which are not yet supported in the above-listed profilers' release versions. Our profiler wraps every OpenMP region into a profiling region and keeps track of each region's exclusive and inclusive metric values. Its position in the code identifies each wrapped region. This allows aligning and comparing regions across variants, even when the number of regions or the execution flow changes. Besides this runtime instrumentation, we support user instrumentation by using pre-defined macros.

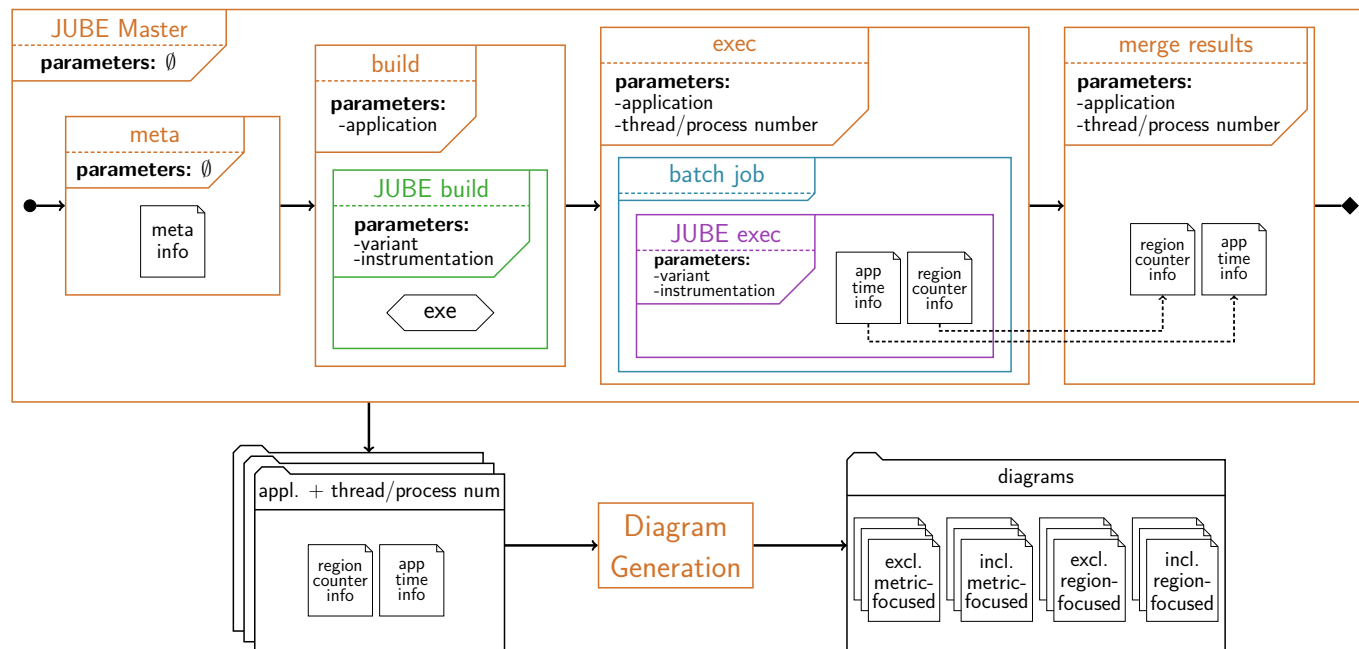


Fig. 1: Overview of our proposed workflow: JUBE controls the execution of the experiments which consists of building the binaries, running them with different configurations and collecting the result. The diagram generator creates PDF files presenting the results

This instrumentation can be used in cases where the code position of a region changes between different variants or when regions apart from the automatically wrapped regions are of interest. An example of this can be seen later in the use case discussion in Section VII.

For the user instrumentation, we do not rely on function calls into the tool library, but we use the OMPT control tool interface instead. Using this interface allows to use the same application for execution with and without the profiler tool attached. If different tools could agree on a common interface for user-instrumentation, defining some additional OMPT values for the control tool functions might avoid changes in the program code for execution with different tools. To enable user instrumentation in C or Fortran codes, we define five new operations in Listing 2. The first three functions allow to register a region name once and then start and stop the region by the returned ID. The last two functions allow to start and stop the region by name. The latter API requires string handling for each region, while the former API helps to avoid the cost. According to the OMPT specification, an OMPT tool can return values larger than 64. Therefore, it would even be possible to fold the start and stop operations into one command. As long as the second argument is smaller or equal to 64, the last argument is interpreted as the name. If the second argument is larger than 64, the last argument is ignored.

We are in the process of upstreaming this lightweight OMPT-based profiler into the LLVM project.

### B. Workflow Using JUBE

As mentioned in Section II-A, JUBE provides an excellent framework for working with larger parameter spaces by branching instances for different parameter settings. However, it only supports branching but no merging of already existing branches. We use nested JUBE calls to gather all data from different branches, with a master JUBE instance that spawns JUBE build and JUBE execute instances. This approach additionally allows the build steps to be parameter-independent from the corresponding execution steps. Every necessary executable only gets built once instead of multiple times.

A single JUBE instance would be sufficient if JUBE would support two features. First, we need the concept of a task barrier or a similar way to model the dependency of each execution step to the build steps of all variants. Second, we need a way to group multiple execution step instances into a single batch job.

Figure 1 depicts an overview of the derived workflow. The master JUBE instance controls the complete workflow and knows the entire parameter space. Starting the master JUBE instance starts a performance experiment. Depending on the use case, one or more *applications* are of interest.

#### Listing 2 User instrumentation API based on OMPT

```
int id = omp_control_tool(region_register, 0, name);
omp_control_tool(region_start_id, id, NULL);
omp_control_tool(region_stop_id, id, NULL);
omp_control_tool(region_start_name, 0, name);
omp_control_tool(region_stop_name, 0, name);
```

Each application is built as *variants*, exchanging some core algorithm and possibly *instrumented* for different analysis tools. For scaling analysis, we executed the applications with varying *thread* or *process* numbers. Different sets of hardware counters can get collected during execution.

The master instance consists of four main steps: meta, build, exec, and merge results. Each step inherits the multi-value parameters of its JUBE instance and the ones from the previous step, but additional parameters can be added. As explained in Section II-A, there is one step instance for every value in the cross-product of all multi-value parameters.

In the *meta* step, all relevant multi-value parameters get written in a metafile, which simplifies the processing of the measured values later.

The *build* step builds all necessary executables for a specified application. There is one build-step instance for every application. Each of them spawns a new *JUBE build* instance inheriting the build step's parameters and additionally gets multi-value parameters for the application variants and instrumentation options. The *JUBE build* instances only consist of one step. Each instance of the step builds an executable corresponding to its current parameter instances.

The *exec* step takes care of the execution for all possible parameter space instances. There is one exec step instance for every combination of application, thread number, and process number. Each instance generates a batch job to be submitted to a cluster. Inside each batch job, a new *JUBE exec* instance is spawned. As for the *JUBE build* instances, there is only one step in the *JUBE exec* instances that inherits the master exec step instance parameters. In addition, the step gets the same multi-value parameters for the application variants and instrumentation options as the *JUBE build* instances. Each instance of the step runs the corresponding executable in an environment fitting to the current parameter instance. Executing all application variants in a single batch job ensures that the variants run on the same hardware allowing us to compare the results. Each step outputs two CSV files, one with runtime information for the complete application variant and one with runtime and counter information for each region in each application variant.

Finally, the *merge results* step collects, for every combination of application and thread/process number, all CSV files from the corresponding JUBE exec instance. For each combination, all measurement data is now stored inside two CSV files but not easily readable.

As mentioned before, we will introduce novel data processing and representation methods in the following section.

## VI. DATA PROCESSING

One of the biggest challenges in comparative performance analysis is the vast amount of data available and required for the analysis. Also, how to present the data clearly and comprehensively. We propose a diagram-based approach, which allows us to visually compare the variant's measurement values. Note that the presented visualizations generated as PDFs should be regarded as mock-ups of the visualizations

and should eventually be integrated into an existing interactive visualization tool. As a first step, we discuss which performance metrics are of interest for the analysis before introducing different diagram types to present them concisely.

### A. Performance Metrics

Base metrics are the simplest type of metrics and refer to metrics measured during the runtime. As we have seen in Section V, our base metrics coincide with the *runtime* and *hardware performance counters*. Apart from base metrics, performance analysis also uses derived metrics, which are, as the name suggests, derived from the base metrics using different arithmetic operations. We consider three different types in this work: relative values of variants, cache usage ratio, and profiler overhead.

1) *Relative variant value*: The relative variant value is the first derived metric used for the comparison of different variants. We determine one variant as the base variant. All metrics get then normalized to the values of the base metrics. For a given metric, variant, region, and thread number, a variant's relative value is defined as the metric value of a measurement of the given variant divided by the base variant's measurement value. By definition, all relative values of the base variant are 1. For this metric, we only consider runs where the profiler is active as the others give no information for single regions. In use cases where the base variant is supposed to be the fastest, the variant's relative value can also be understood as the overhead of the variant. An example use case would be data race analysis, where the base variant is executed without an analysis tool.

2) *Cache usage ratio*: The second metric to compare variants is the cache usage ratio. For hardware counters that measure cache hits at the different cache levels and the main memory, a more relevant metric than the relative value of a variant is the ratio with which each cache level is hit for a given variant, region, and thread number. The cache usage ratio of a cache is defined as the cache-hit value of this cache level divided by all memory requests. The cache-hit ratios of all caches and main memory together add up to 1 and describe a cache-usage distribution to compare between different application variants. Other tools suggest looking at memory and cache bandwidths to identify bottlenecks, which we find less relevant for the comparative performance analysis.

3) *Profiler overhead*: The profiler overhead, in contrast to the other two metrics, only gets calculated to verify a measurement run's usability and not for the actual comparison. The presence of a profiler in the execution of an application can prolong and distort the runtime. To verify that this effect is negligible, we consider the profiler overhead. Note that the overhead will depend on the chosen tool and measurement configuration. For a given metric, variant, region, and thread number, we define the profiler overhead as the metric value of a configuration run with a profiler divided by the value of the same configuration run without the profiler. While the other two metrics are displayed using diagrams, our

workflow checks for profiler overheads higher than a pre-defined threshold and prints a warning for all measurements that fail the check. In this case the tool user can decide whether the increased overhead is acceptable or whether the execution should be repeated to rule out an outlier.

## B. Data Presentation

Variant overhead and cache usage ratio are defined over a 4-dimensional parameter space with the dimensions metric, variant, region, and thread or process number. Just comparing each value by hand is therefore infeasible. We propose a diagram-based approach to display the data more clearly and understandably by reducing the parameter space to a two-dimensional diagram space where each diagram depicts two dimensions ( $dim_1, dim_2$ ) at once. Comparing two of the proposed diagrams equals the comparison of  $2 \cdot |dim_1| \cdot |dim_2|$  standard value diagrams, thus reducing the number of needed manual comparisons significantly and making a user-driven analysis and comparison feasible.

A diagram always showcases all thread numbers by using different data series. This allows us to get a quick overview of the scaling behavior. On the other hand, a diagram never displays values for more than one variant at a time. We argue that, of all four dimensions, the variant number is the smallest and, therefore, the most feasible to compare by hand between different diagrams. For the remaining two dimensions, it is unclear which one should be fixed for a diagram, so we introduce two different types of diagrams in the following. The expressiveness of the two different types depends on the use case.

1) *Metric-focused diagrams*: Metric-focused diagrams fix the examined region and plot values for different metrics on the x-axis. We generate a bar diagram for the left y-axis, where the bars display different metric's relative variant values (see Section VI-A). There are multiple bars for each entry on the x-axis, with each bar corresponding to a different number of threads. The absolute metric values are shown on the right y-axis and are displayed as a dot plot over the bar diagram. A plus represents the absolute metric value of the base variant for the number of threads of the corresponding bar. Similarly, the cross represents the absolute value of the chosen variant. An example for this type of diagram can be seen in Figure 2.

To display the cache-usage ratio, we choose stacked ring diagrams instead of bar diagrams as we found them easier to interpret and compare. An example is shown in Figure 3 on page 8. Each ring corresponds to a different thread count and displays the different cache hit ratios. The 12.5% mark for L1 misses highlights a memory-access pattern that uses all eight double elements out of a cache line with eight values. If more memory accesses are served from higher cache levels, this might indicate a bad cache reuse. However, it could also be the result of larger memory accesses using vector instructions. Since cache ratio does not contain information about the total number of memory accesses, this diagram must always be seen in the context of the absolute metric values.

2) *Region-focused diagrams*: Region-focused diagrams fix the examined metric and plot the values of different regions on the x-axis, as can be seen in Figure 4 on page 9. Again, we use bar graphs for the relative values and overlay dot plots for the absolute values. The left y-axis displays the relative values, and the right y-axis the absolute values. Absolute values in region-focused diagrams have the additional advantage that the user can determine each region's importance. A region with a high overhead but a very short runtime compared to all the other regions is often not of interest. The absolute values are aggregated over all threads. An OpenMP initial task region (see `InitTaskRegion` in the diagram) represents the execution time of the initial thread. Similarly, if a user-defined region starts in serial code, it only tracks the execution time and performance counters of the initial thread. `InitTaskRegion` and user-defined regions are the only regions expected to scale down for increasing thread counts in a strong scaling experiment.

## C. Analysis Workflow: From Overview to Details

When studying multiple applications, an overview diagram hints at problematic applications or application versions with the highest runtime overhead caused by code changes or by an analysis tool. Starting from there, we can look at the individual application and inspect the overhead respectively performance change at a per region level as in Figure 4. The implicit task regions in this figure correspond to different OpenMP parallel regions in the application code or in linked libraries. We identify the region of interest, either the region with the highest relative overhead or the region with the highest absolute runtime in the variant. Finally, we inspect the collected hardware performance counter for the region of interest and correlate spiking relative counter values with the observed time overhead. A graphical user interface like CUBE could support this workflow by presenting the different views and allowing to interactively browse through the data.

## VII. USE CASES

In order to demonstrate the applicability of our workflow, we apply it to two selected mini-apps that we extracted from real application codes for analysis purposes. The first mini-app is based on the `CalculiX2` application. `CalculiX` is an open-source software package developed to solve three-dimensional field problems in Finite Element Analysis and Computational Fluid Dynamics by implementing the finite element method and the finite volume method. In previous work, we analyzed a complex kind of load imbalance [18] that we found in the main computational kernel of the `CalculiX` application, where it uses the `GMRES` method [19] in parallel to solve large, sparse, asymmetric linear systems. Our mini-app features this parallel `GMRES` kernel, and we developed different variants by exchanging the `GMRES` implementation in use. The base variant uses a `GMRES` implementation from the `SLATEC`

<sup>2</sup><http://www.calculix.de/>

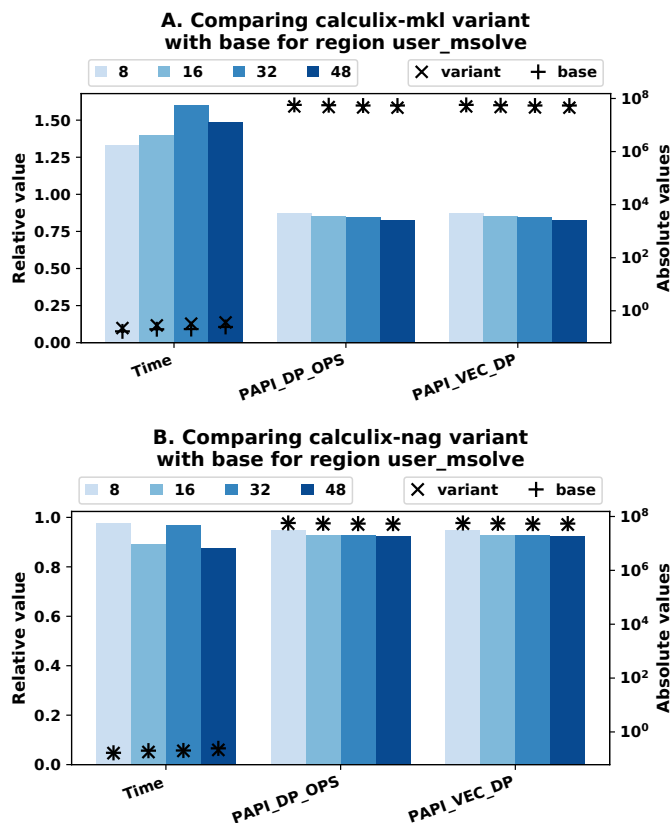


Fig. 2: Metric-focused diagrams for the MKL and NAG variants. The bars correspond to the left y-axis and display the values relative to a base-line variant. The absolute values are overlaid as a dot plot corresponding to the right y-axis.

project<sup>3</sup> similar to the original CalculiX application. Additionally, we integrated the reverse communication interface-based GMRES implementations of the Intel Math Kernel Library and the NAG Library into our mini-app, which we refer to as the MKL or NAG variant, respectively.

The second mini-app is based on the KKRhost code of the Juelich KKR suite<sup>4</sup> which contains a variety of density functional theory codes all based on the Korrington-Kohn-Rostoker Green function method. The *KKRhost* code is a hybrid MPI + OpenMP parallelized application [20]. Our mini-app is an OpenMP-only application focusing on the local computations that happen inside each MPI rank during the solution of the algebraic Dyson equation with respect to the full *KKRhost* code. In order to solve the algebraic Dyson equation, a k-point integration loop is performed. The scattering path operator is computed for each discrete k-point in the simulation domain, requiring an LU decomposition to work around a theoretical matrix inversion. For this mini-app, we developed two different variants with different parallelization schemes. In the first variant, only the computations of the LU decompositions happen in parallel by calling into the threaded version of the Intel Math Kernel Library (MKL) while all other computations

are performed serially in the main thread of the process. In the second variant, the loop over the set of k-points is parallelized using OpenMP work-sharing constructs. The serial version of MKL is called for the LU decompositions by each OpenMP thread instead.

In the following, we compare the different variants of the two mini-apps in terms of their performance to showcase the usability of our workflow. A next step would be to use our framework to verify that the performance behavior observed for the variants will stay the same when applying the changes to the original code. However, this step exceeds the bounds of this use case study.

### A. Experimental setup

We run the experiments on dual-socket nodes of Claix-18, comprising two 24 core Intel Skylake Platinum 8160 CPUs and 192GB memory. The nodes have hyper-threading disabled and sub-NUMA clustering enabled. For the execution, we export `OMP_PLACES=cores` and `OMP_PROC_BIND=close` to bind the thread to individual cores. Due to a bug in the OpenMP runtime, even with close binding, the threads might end up distributed between the sub-NUMA domains of a socket. The operating system is CentOS 7.9.

### B. CalculiX study

In the following, we apply our analysis workflow to three variants of our CalculiX mini-app to gain insights into performance differences between the different GMRES implementations. We do not want to understand this as a benchmark of the different GMRES implementations, but more a performance verification of the integration of the different GMRES implementations into our mini-app. In order to emphasize the performance insights that our proposed diagrams can give, we focus on the evaluation of results obtained for the code region, which applies a Jacobi preconditioner in GMRES. For alignment, we marked this region with user instrumentation. We refer to it in the following as *msolve*.

1) *Relative metrics analysis*: Figure 2 shows relative metric values for time, double-precision FLOP and the number of double-precision vector instructions. As a side note, we can see that the *msolve* region is not vectorized in all variants, but we could determine that with traditional tools and workflows.

Figure 2a depicts the metric values for the MKL variant of our mini-app. First, we recognize that the time spent in the *msolve* region is roughly between 25% and 60% higher compared to the base variant of our mini-app. At the same time, the hardware counters related to double-precision floating-point operations reveal that we have only about 75% compared to the base variant. This means we need further analysis to investigate the slowdown of this region.

Figure 2b shows the metric values obtained with the NAG variant of our mini-app. For 8 and 32 threads, the runtime of the *msolve* region is only slightly faster than in the base variant. However, with 16 and 48 threads, the code region is executed more than 10% faster. Looking at the hardware

<sup>3</sup><https://www.netlib.org/slatec/lin/>

<sup>4</sup><https://jukkr.fz-juelich.de>

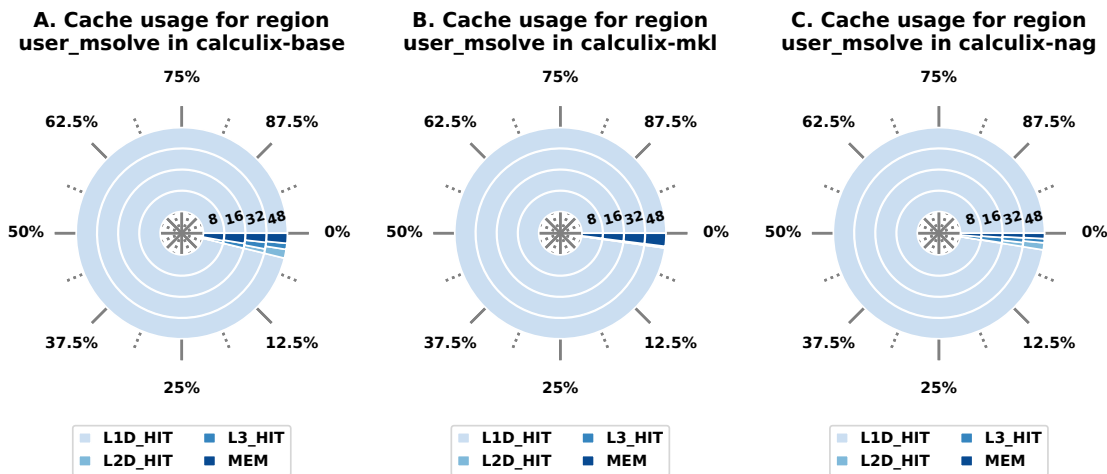


Fig. 3: Diagrams displaying cache usage ratios for base, MKL and NAG variants. The rings represent different thread counts.

counters, we see that in all cases, roughly 10% less double-precision FLOP are executed. Since this only matches the speedup for 16 and 48 threads but not for 8 and 32 threads it is unlikely that the reduced number of FLOP and instructions is the reason for this speedup.

2) *Cache usage ratio analysis:* For both the MKL and the NAG variant, the hardware counters on FLOP and instructions are insufficient to explain the runtime difference compared to the base variant. Hence, we next consult the cache usage distribution for the three variants shown in Figure 3. From the innermost to the outermost ring, the thread count increases from 8 to 48.

Figure 3a shows the cache usage distribution for the msolve region in the base variant. Our proposed ring depiction’s main advantage is the ability to quickly recognize how data is reused from the different levels of caches between the main memory and the L1 cache in one view for multiple thread counts. Comparing the ring plots for the different variants allows identifying relevant changes in cache usage. We see that increasing the number of threads leads to better usage of the L2 cache in this application case, as we would expect for a strong scaling experiment. The main memory contribution continuously shrinks while the contribution from the L2 cache grows and becomes increasingly more visible in the diagram. This behavior is reasonable as the individual problem size on each thread shrinks if the thread number grows.

Figure 3b shows the cache usage distribution for the MKL variant. Remember that we noticed a slowdown for msolve between 25% and 60% before. If we compare the cache utilization to the base variant, we see that the reuse from L1 cache increased from 96% to roughly 98%. However, at the same time, nearly all of the L1 misses are satisfied from memory across all thread counts. Consulting Figure 4a, we can see that this variant performs about 50% more read accesses, which explains the decrease in the L1 cache miss ratio. Thus, the runtime overhead of more than 50% in the MKL variant for these thread counts seems to be mainly caused by fetching data from main memory rather than from L2 or L3 cache.

Overall, the MKL variant does not reuse the L2 cache, and there is also almost no reuse of the L3 cache compared to the base variant.

Figure 3c shows the NAG variant’s cache usage distribution. Again, Figure 4b helps to explain the decrease in the L1 cache miss ratio. This variant showed a 10% shorter runtime for the msolve region for 16 and 48 threads. Comparing the cache usage distribution with the base variant, the reuse on the different cache levels looks somewhat similar at first. However, there is one important exception: we notice that the main memory’s contribution has significantly reduced for all thread counts. For 16 threads, this change is more significant and favors better reuse from the L3 cache. For 48 threads, both the contribution from main memory and L3 cache shrink in favor of better reuse from L2 and L3 cache.

3) *Per region comparison:* Figure 4 highlights another interesting issue of our MKL variant. This variant performs double the memory accesses inside the matvec routine compared to the other variants. This pattern is consistent with other metrics such as execution time or FLOP. Reviewing the implementation of the reverse communication GMRES, we could identify the second matvec call being caused by the residuum calculation. Once this issue was identified, we could use a different MKL API to avoid the second matvec call.

### C. JuKKR study

In the following, we apply our analysis workflow to two variants of our JuKKR-KKRhost mini-app to gain insights into performance differences between the different implemented parallelization schemes.

1) *Relative metrics analysis:* Figure 5 highlights the scalability of the OpenMP variant across most performance counters. At the same time, the diagram also shows that relying on parallel MKL routines for the LU decomposition does not significantly impact any of the metrics. Since we compare two completely different parallelization approaches, we can place the user region only in the serial code. This highlights a limitation of the current state of our profiler, and we would

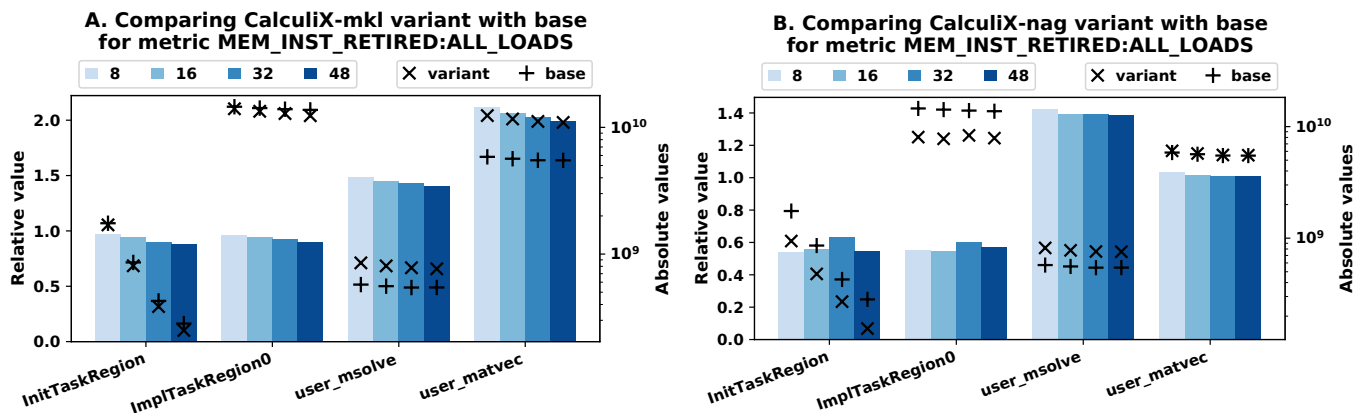


Fig. 4: Region-focused diagrams with relative values for retired load instructions of the MKL and NAG variants. The absolute values are overlaid as a dot plot corresponding to the right y-axis.

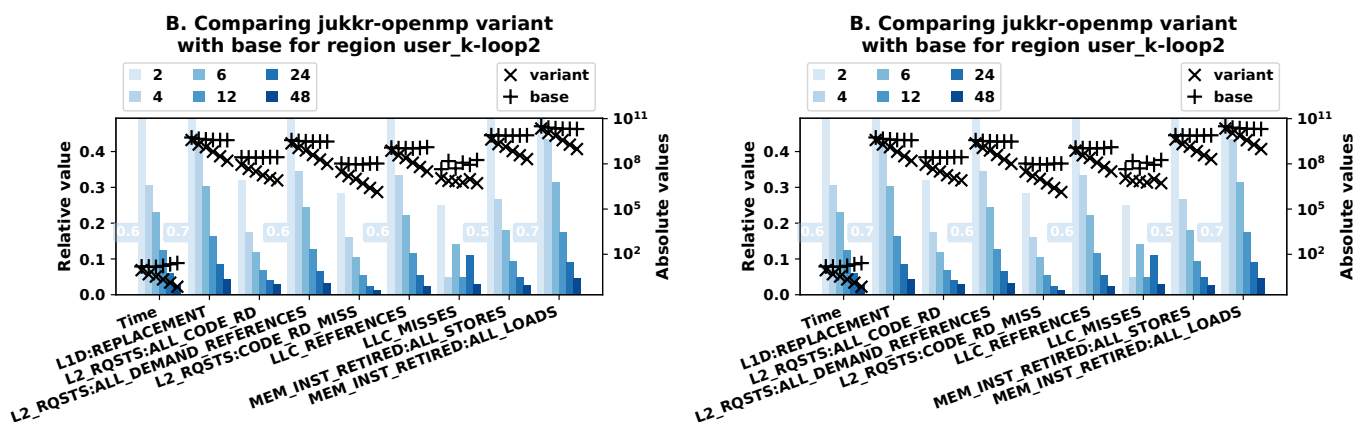


Fig. 5: Metric-focused diagrams with cycle and cache count values for the k-loop2 region of the JuKKR-KKRhost mini-app. Since the user-defined region only measures the primary thread, the execution time of the openmp variant shows the expected strong scaling behavior

gain more insight if user-defined regions would include the counter values of threads forked inside of the region. Looking at the overall performance counters shows that the base version

does not execute significantly more instructions of memory operations. The base version mainly keeps the initial thread busy, while most of the other threads are idling.

2) *Cache usage ratio analysis:* Figure 6 shows that both variants roughly have the same reuse of L1 cache. However, the OpenMP variant shows better L2 reuse and fewer accesses into the L3 cache and main memory. Again, the data might be a bit distorted because we only track accesses on the initial thread.

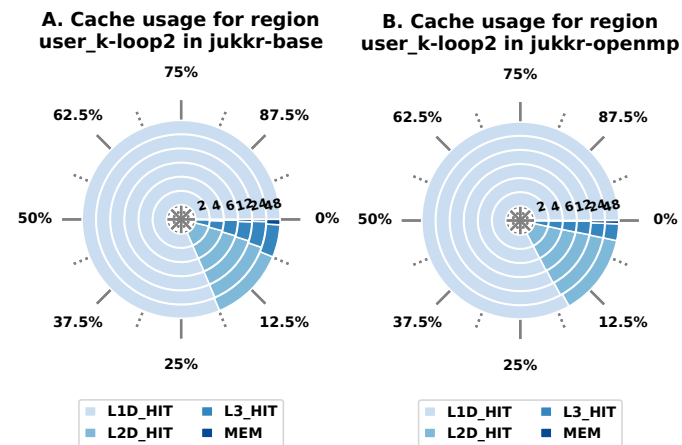


Fig. 6: The two variants of the JuKKR-KKRhost mini-app show a similar cache reuse ratio with slightly fewer accesses into L3 cache and main memory

## VIII. CONCLUSIONS

This paper presented a novel workflow for comparative performance analysis to better understand how algorithmic changes may impact an application’s performance. While using existing performance analysis tools to gather the relevant performance data, our presented workflow overcomes difficulties with such tools when algorithmic changes or varying running conditions are the main interest of investigation. We showed how these parametric changes could be handled by nesting several instances of the JUElich Benchmarking Environment (JUBE) framework. Alongside our presented

workflow, we proposed different diagrams that allow for a comprehensive analysis of a vast amount of data in a four-dimensional space of metrics, application variants, code regions, and thread or process numbers. We proposed metric-focused diagrams that help understand how relative and absolute values of metrics develop over different numbers of threads or processes for a fixed variant and region. Our proposed visualization of the cache usage distribution as ring diagrams, on the one hand, gives an idea of how an application reuses the different cache levels between main memory and L1 cache for a fixed variant and a fixed region. On the other hand, this depiction enables comparisons of the same region between different variants to relate speedup or slowdown with better respectively worse cache utilization. Region-focused diagrams help to identify interesting regions by fixing the variant and metric dimensions. The absolute values shown in these diagrams are helpful to give relative values such as the cache-hit ratio more context. Finally, we showed how our proposed workflow could be applied to a typical use case, such as comparing different implementations of linear solvers by evaluating different GMRES implementations on our CalculiX mini-app. As a second use-case, we compared two different parallelization strategies for the k-point integration loop in JuKKR-KKRhost.

We plan to improve the exclusive measurement of metrics for nested regions such as recursive tasks for future research. Furthermore, inclusive measurement of user-defined regions should include the metrics for threads forked within the region. To get more insight into performance behavior, we will investigate explicitly separating the time spent in the parallel runtime from time spent in application code. Similarly, we will look into statistics on per-thread rather than aggregated profile data, which will give insights into load imbalances. Furthermore, we will investigate interfacing with LIKWID, as our predefined counter group to calculate the cache-hit ratio is currently limited to the Skylake architecture. Finally, while we used the JUBE framework with a specific performance tool, the presented workflow itself is agnostic to specific tools and thus could easily be ported to using other tools and potentially focus on different measurement information. Moreover, existing performance tools could be extended to natively incorporate the workflow and analysis techniques presented in this work.

#### ACKNOWLEDGEMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 824080.

#### REFERENCES

[1] S. Lührs, D. Rohe, A. Schnurpfeil, K. Thust, and W. Frings, "Flexible and Generic Workflow Management," in *Parallel Computing: On the Road to Exascale*, ser. Advances in parallel computing, vol. 27. International Conference on Parallel Computing 2015, Sep 2016, pp. 431–438.

[2] J. Treibig, G. Hager, and G. Wellein, "Likwid: Lightweight performance tools," in *Competence in High Performance Computing 2010*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds. Springer Berlin Heidelberg, 2012, pp. 165–175.

[3] D. Terpstra, H. Jagode, H. You, and J. Dongarra, *Collecting performance data with PAPI-C*, 05 2010, pp. 157–173.

[4] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, 2012, pp. 79–91.

[5] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice & Experience - Scalable Tools for High-End Computing*, pp. 702–719, 2010.

[6] S. Shende, "The Tau Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, pp. 287–311, 2006.

[7] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool-Set," in *Tools for High Performance Computing*, 2008, pp. 139–155.

[8] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube v4: From performance report explorer to performance analysis tool," *Procedia Computer Science*, vol. 51, pp. 1343–1352, 2015, international Conference On Computational Science, ICCS 2015.

[9] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore, "An algebra for cross-experiment performance analysis," in *Proc. of the International Conference on Parallel Processing (ICPP), Montreal, Canada*. IEEE Society, August 2004, pp. 63–72.

[10] M. Weber, R. Brendel, T. Hilbrich, K. Mohror, M. Schulz, and H. Brunst, "Structural clustering: A new approach to support performance analysis at scale," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, 2016, pp. 484–493.

[11] A. Calotou, "Automatic Empirical Performance Modeling of Parallel Programs," Ph.D. dissertation, Technische Universität, Darmstadt, 2018.

[12] C. Coarfa, J. M. Mellor-Crummey, N. Froyd, and Y. Dotsenko, "Scalability analysis of SPMD codes using expectations," in *ICS*. ACM, 2007, pp. 13–22.

[13] J. R. Madsen, M. G. Awan, H. Brunie, J. Deslippe, R. Gayatri, L. Oliker, Y. Wang, C. Yang, and S. Williams, "Timemory: Modular Performance Analysis for HPC," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds. Springer International Publishing, 2020, pp. 434–452.

[14] A. Batele, S. Brink, and T. Gambin, "Hatchet: pruning the overgrowth in parallel profiles," in *SC*. ACM, 2019, pp. 20:1–20:21.

[15] K. Mohror and K. L. Karavanic, "Evaluating similarity-based trace reduction techniques for scalable performance analysis," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*, 2009.

[16] V. Pillet, V. Pillet, J. Labarta, T. Cortes, T. Cortes, S. Girona, S. Girona, and D. D. D. Computadors, "Paraver: A tool to visualize and analyze parallel code," In WoTUG-18, Tech. Rep., 1995.

[17] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT and OMPD: OpenMP tools application programming interfaces for performance analysis and debugging," in *International Workshop on OpenMP (IWOMP 2013)*, 2013.

[18] F. Orland and C. Terboven, "A Case Study on Addressing Complex Load Imbalance in OpenMP," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, K. Milfeld, B. R. de Supinski, L. Koesterke, and J. Klinkenberg, Eds. Springer International Publishing, 2020, pp. 130–145.

[19] Y. Saad and M. H. Schultz, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM J. Sci. Stat. Comput.*, vol. 7, no. 3, p. 856–869, Jul. 1986.

[20] P. Rübmann, "Spin scattering of topologically protected electrons at defects," Dissertation, RWTH Aachen University, Jülich, 2018, druckausgabe: 2018. - Onlineausgabe: 2018. - Auch veröffentlicht auf dem Publikationsserver der RWTH Aachen University; Dissertation, RWTH Aachen University, 2018.