

# PAP: Performance Analysis Portal for HPC Applications

Master Thesis

University of Basel  
Faculty of Science  
Department of Mathematics  
and Computer Science

Examiner: Prof. Dr. Florina M. Ciorba  
Supervisor: Jonas H. Müller Korndörfer

Thomas Jakobsche  
thomas.jakobsche@stud.unibas.ch

14.02.2020



## **Abstract**

Current HPC system architects strive for more performance by relying on larger numbers of processing units that work in parallel, instead of further developing the speed of an individual processing unit. In order to exploit the parallelism provided by these HPC systems, scientific applications increasingly use parallel programming paradigms. Performance analysis of these parallel applications can help to solve large-scale scientific problems more efficiently. We identified several shortcomings and possible improvements in the context of performance analysis: (a) vague and inconsistent analysis methodologies, (b) manual selection of applications for performance analysis, and (c) high-level comparison of multiple application profiles. In this master thesis we present the following solutions to the aforementioned problems: (a) representative performance metrics based on data collected by Score-P and a semi-automatic analysis workflow, embedded in a complete analysis methodology, (b) an application database that stores performance data and supports querying, and (c) performance analysis of individual and multiple applications, investigation of programming paradigm usage (MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC), and application similarity grouping based on k-means clustering. The proposed solutions are components of PAP: Performance Analysis Portal for HPC Applications, a web based portal for performance analysis of parallel applications.

# Contents

<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>4</b>
1.1 High Performance Computing . . . . .	4
1.2 Performance Analysis . . . . .	5
1.3 Motivation - Stating the Problem . . . . .	6
1.3.1 Summarising the Problem . . . . .	6
1.4 Goals - Planning the Solution . . . . .	7
1.4.1 Summarising the Solution . . . . .	7
1.5 The Performance Analysis Portal . . . . .	8
<b>2 Related Work and Competitors</b>	<b>10</b>
2.1 Performance Analysis Frameworks and Tools . . . . .	10
2.1.1 Score-P . . . . .	11
2.1.2 TAU . . . . .	12
2.1.3 Scalasca . . . . .	14
2.1.4 Vampir . . . . .	16
2.1.5 HPCToolkit . . . . .	19
2.1.6 Paraver . . . . .	22
2.1.7 mpiP . . . . .	23
2.1.8 Overview and Comparison . . . . .	26
2.2 Methodologies and Characteristics . . . . .	27
2.2.1 Oxbow and PADS . . . . .	27
2.2.2 A Large-Scale Study of MPI Usage . . . . .	32
2.2.3 Benchmark Similarity . . . . .	34
2.2.4 Empirical Performance Evaluation . . . . .	37
2.2.5 Communication Patterns . . . . .	39
<b>3 Proposed Methodology</b>	<b>45</b>
3.1 Server Side - Data Access Layer . . . . .	45
3.1.1 Node.js - JavaScript Runtime Environment . . . . .	45
3.1.2 MongoDB - Document Oriented NoSQL Database . . . . .	46
3.1.3 PHP - Server Management . . . . .	46
3.2 Client Side - Presentation Layer . . . . .	46
3.2.1 HTML - Hypertext Markup Language . . . . .	46
3.2.2 CSS - Cascading Style Sheets . . . . .	47
3.2.3 JavaScript - Scripting Language . . . . .	47
3.2.4 Google HTML/CSS Style Guide . . . . .	47

3.3	Additional Third-Party Libraries . . . . .	48
3.3.1	jQuery - JavaScript Library . . . . .	48
3.3.2	Plotly.js - Graphing Library . . . . .	48
3.3.3	Mask.js - Input Masking . . . . .	48
3.3.4	Simple Statistics - Statistical Methods . . . . .	48
<b>4</b>	<b>Design and Development</b>	<b>49</b>
4.1	Server Side - Decisions and Development . . . . .	49
4.1.1	Database Interaction . . . . .	49
4.1.2	Computation on the Client . . . . .	49
4.1.3	PHP - Command Line Access . . . . .	50
4.1.4	Dealing with JavaScript Code Injection . . . . .	50
4.2	Client Side - Preparation and Measurement . . . . .	50
4.2.1	Performance Data Parsing . . . . .	51
4.2.2	Score-P Function Groups . . . . .	52
4.2.3	SLURM Job Script Generator . . . . .	52
4.3	Client Side - The Application Database . . . . .	52
4.3.1	Application Database Structure . . . . .	53
4.3.2	Metadata - Identifying Database Entries . . . . .	53
4.3.3	Additional Characteristics and Input Fields . . . . .	54
4.3.4	Data Upload and Input Form Masking . . . . .	55
4.3.5	Programming Paradigm Assignment . . . . .	56
4.3.6	Filtering and Scope Selection . . . . .	57
4.4	Client Side - Analysis and Visualisation . . . . .	57
4.4.1	Application Group Summary . . . . .	58
4.4.2	Application Region List . . . . .	59
4.4.3	Application Group Comparison . . . . .	60
4.4.4	Programming Paradigm Statistics . . . . .	61
4.4.5	K-Means Similarity Clustering . . . . .	62
<b>5</b>	<b>Analysis Methodology and Workflow</b>	<b>63</b>
5.1	Introducing the Analysis Methodology of PAP . . . . .	63
5.1.1	Workflow Steps - Overview . . . . .	64
5.2	Step 1: Preparation and Measurement . . . . .	65
5.2.1	Performance Data Collection with Score-P . . . . .	65
5.2.2	SLURM Job Script Generator . . . . .	66
5.2.3	Extracting Data from a Score-P Profile . . . . .	67
5.3	Step 2: Application Database Interaction . . . . .	68
5.3.1	Add and Update Database Entries . . . . .	68
5.3.2	Export Database Entries . . . . .	73
5.3.3	Listing and Querying Database Entries . . . . .	74
5.3.4	Removing Database Entries . . . . .	76
5.4	Step 3: Performance Analysis . . . . .	77
5.4.1	Analysis Scope Selection . . . . .	77
5.4.2	Individual Application Summary . . . . .	78
5.4.3	Application Comparison and Scaling . . . . .	80
5.4.4	Programming Paradigm Statistics . . . . .	81
5.4.5	Application Similarity Clustering . . . . .	82

<b>6</b>	<b>Discussion</b>	<b>83</b>
6.1	Measuring Success . . . . .	84
6.1.1	Comparison of Performance Metrics . . . . .	84
6.1.2	Differences in Sub Groups of MPI . . . . .	85
6.1.3	K-Means vs. Hierarchical Clustering . . . . .	85
6.1.4	Comparison of Programming Paradigm Statistics . . . . .	86
6.1.5	Other Web-Based Application Databases . . . . .	86
6.2	Extensibility and Future Work . . . . .	87
6.2.1	Addressing Limitations . . . . .	87
6.2.2	Further Development . . . . .	87
	<b>References</b>	<b>91</b>

# Chapter 1

## Introduction

This chapter contains an introduction into the scientific background and the context of this master thesis. It also contains the motivation, goals, and a summary of the presented work at the end of the chapter.

### 1.1 High Performance Computing

Scientific applications need high computing power provided by high performance computing (HPC) systems in order to solve large-scale problems [15]. Instead of further developing the speed of an individual processing unit, current HPC system architects strive for more performance by relying on larger numbers of processing units that work in parallel. Increasing power dissipation and little room for improvement of instruction-level parallelism are some of the reasons for this strategy [14].

Processing units of parallel architectures solve problems by distributing the work and performing parallel computation. It is necessary for scientific applications to adopt the higher degrees of parallelism in current architectures, to expand their potential and efficiency. It is an ongoing task to close the gap between sustainable production performance and peak performance achieved by HPC systems in the face of increasing numbers of processing units [30].

There are different strategies and programming paradigms for applications in order to exploit the parallelism provided by HPC systems. The most prominent approaches are MPI [15] and OpenMP [11]. Generally, MPI (Message Passing Interface) is a method to program on distributed memory devices: parallel processes are working in their individual memory space and exchange messages to share data. OpenMP (Open Multi-Processing) is a method to program on shared memory devices: parallel threads have access to the same data. Furthermore hybrid approaches are possible, where MPI handles inter-node and OpenMP intra-node parallelism.

## 1.2 Performance Analysis

Application developers try to achieve high performance by fully utilizing the capabilities provided by programming paradigms and HPC systems. Performance analysis tools enable the investigation and analysis of representative metrics and characteristics. These tools help the user to investigate application performance behaviour and find performance bottlenecks. There are several different performance analysis tools which vary in the capabilities that they offer, but also in the programming languages and paradigms they support. Therefore, a single tool is frequently not enough for the analysis of complex parallel applications.

The performance of parallel applications can also provide information about the system they are running on. Evaluating the performance of HPC systems can be achieved with dedicated performance evaluation applications (benchmarks), that are often derived from real scientific applications in order to mimic their parallel execution behaviour. By supporting the analysis and comparison of performance characteristics, benchmarks also support the development and improvement of applications and HPC systems. Therefore, they play a crucial role in the investigation of different architecture approaches and programming paradigms.

Although necessary for the development of a new HPC system, benchmarking with real scientific applications proved to be impractical because they are limited by their own complexity [17]. Real applications are usually complicated and have many requirements, therefore they are not suited for the evaluation of HPC systems in an early development stage. Even with requirements met, there is still major effort associated with the porting of a large program to a new architecture [8].

Given an increasing number of scientific applications, benchmarks and performance analysis tools, the decision on an appropriate performance analysis approach becomes increasingly difficult. While scientific studies often state the experimental setup, tools, and performance metrics, they seldom provide the workflow for their analysis. This becomes a problem when studies are to be replicated or extended. An analysis workflow provides a step-by-step guide on how to setup the experiment, collect metrics, and analyse performance data. The analysis workflow together with the metrics form a methodology, which is a representation of the complete analysis process. A good methodology can provide orientation in the landscape of performance analysis.

## 1.3 Motivation - Stating the Problem

The general motivation for the presented work, is to gain better understanding of programming paradigm usage in order to analyse parallel application performance. We investigated the overall landscape of performance analysis, and how other studies approach this issue. We identified several shortcomings and possible improvements.

### Vague and Inconsistent Analysis Methodologies

We noticed that a lot of work follows different analysis workflows with different performance metrics. This becomes a problem for researchers that want to gather and compare performance results about a high number of applications. Another problem arises when the analysis methodology is not fully stated, and work needs to be replicated. This calls for an analysis methodology, that clearly states performance metrics and the corresponding analysis workflow.

### Manual Selection of Applications for Performance Analysis

In order to select one or more candidates for experiments and performance optimization, researchers need to compare a high number of applications. The problem is that manual selection and comparison of candidates is a Sisyphean task in the face of an increasing number of scientific applications and benchmarks. Approaches that organise performance data and provide dynamic filtering thereof, can support the selection process of candidate applications for performance analysis.

### High-Level Comparison of Multiple Application Profiles

Comparison of multiple applications has been mentioned as part of the problem in preceding paragraphs. When analysing a single application, investigation of the event trace can provide very detailed insight into individual performance behaviour. However, researchers that want to compare multiple applications can also benefit from approaches that provide high-level comparison of aggregated performance data from application profiles.

#### 1.3.1 Summarising the Problem

The following list summarises the shortcomings and possible improvements that we want to address with the work presented in the subsequent Chapters:

- (a) Vague and inconsistent analysis methodologies (different performance metrics, incomplete analysis workflows).
- (b) Manual selection of applications for performance analysis (requiring dynamic filtering and grouping of applications).
- (c) High-level comparison of multiple application profiles (high-level comparison instead of in-depth single analysis).



## 1.4 Goals - Planning the Solution

Our goals are rooted in our experiences with the related work in performance analysis, summarised in the preceding section. The overarching goal is a performance analysis portal that provides: a complete analysis methodology, a way of organising and filtering performance data, comparative performance analysis, and similarity grouping of multiple applications.

### **Analysis Methodology with Semi-Automatic Workflow**

We aim for a transparent analysis methodology that supports the user with an analysis workflow that guides from preparation to measurement, execution, performance data collection and finally performance analysis. The portal should provide instructions and functionality embedded in a semi-automatic workflow with graphical user interface.

### **Application Catalogue for Performance Data**

As part of the methodology we also define a list of representative performance metrics and characteristics that help to investigate parallel performance behaviour. The organisation of performance data shall be in the form of a catalogue that stores information and characteristics of multiple applications. Entries should be distinguished by version, and other suitable metadata. The catalogue should support dynamic filtering and querying of entries.

### **Performance Analysis and Application Similarity**

The user shall be able to compare all applications inside the catalogue with different visualisation methods. The focus should be on profile comparison of multiple applications and application similarity grouping based on different characteristics.

#### 1.4.1 Summarising the Solution

The following list summarises the solutions that we developed to address the problems mentioned in the preceding Section 1.3:

- (a) Analysis methodology with semi-automatic workflow (list of performance metrics, analysis workflow embedded in graphical user interface).
- (b) Application catalogue for performance data (extensible application catalogue that supports filtering and querying of entries).
- (c) Performance analysis and application similarity (comparison of multiple application profiles and grouping of applications based on similarity).

## 1.5 The Performance Analysis Portal

The result of this master thesis is a web-based Performance Analysis Portal for HPC Applications (PAP). It was designed to fulfil the aforementioned goals and follows a client-server architecture. Figure 1.1 gives an overview of the portal and the technologies we used to develop it.

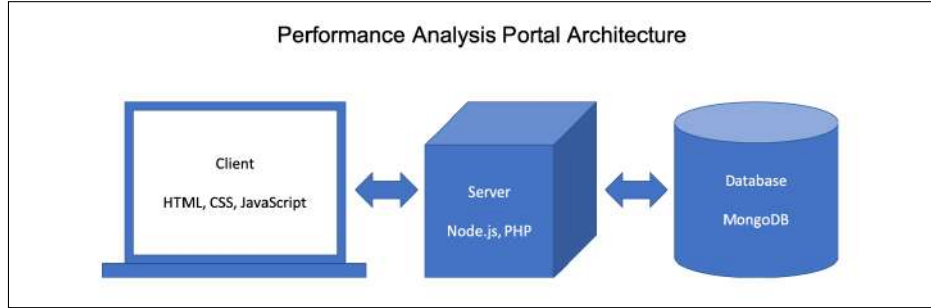


Figure 1.1: The Client - Server Architecture of PAP: Performance Analysis Portal for HPC Applications, showing the technologies we used for each component.

The client side presents itself as a graphical user interface written in HTML, CSS and JavaScript. The server side is build upon the Node.js JavaScript runtime environment, it provides access to the application database that was developed with the document-oriented database program MongoDB.

The portal provides a semi-automatic performance analysis workflow with step-by-step instructions on (1) Preparation & Measurement, (2) Database Access, and (3) Performance Analysis. The preparation and measurement phase relies on the profiling capabilities of Score-P for performance data collection. The portal also offers a SLURM job script generator for convenience. Figure 1.2 gives an overview of the components of step 1: Preparation & Measurement.

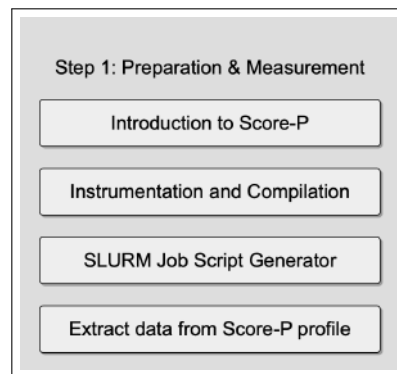


Figure 1.2: Step 1 of our performance analysis workflow. The Preparation & Measurement step relies on Score-P for instrumentation and compilation of parallel applications, in order to generate aggregated profile data. This step also provides a SLURM job script generator.

The application database is capable of storing multiple different metrics, either uploaded in the form of a Score-P generated profile or manually entered by the user. With the option of manually entering performance data, the user can also work outside of the Score-P infrastructure. Figure 1.3 gives an overview of the components of step 2: Database Access.

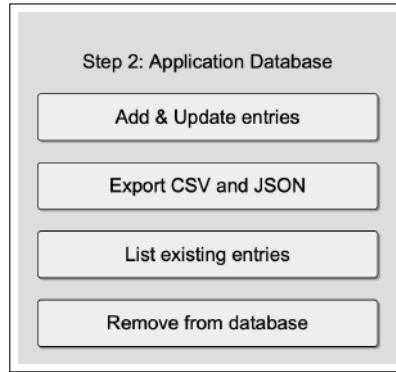


Figure 1.3: Step 2 of our performance analysis workflow. The Application Database supports the upload of profile data generated in step 1, but it also accepts manual insertion of performance data. The user can query the database to filter for specific applications and export the contents of database entries in various formats.

The analysis and visualisation phase offers functionality for the investigation of individual applications, comparing multiple applications, generating statistics about programming paradigms, and grouping applications with k-means clustering. Figure 1.4 gives an overview of the components of step 3: Performance Analysis.

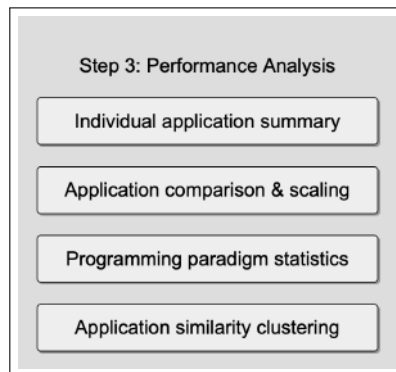


Figure 1.4: Step 3 of our performance analysis workflow. The Performance Analysis step presents itself as a series of individual and multiple comparison views, that provide the user with a number of plots and tables about applications, as well as programming paradigms.

## Chapter 2

# Related Work and Competitors

The following chapter explores related work in the context of parallel applications performance analysis. We focus on performance analysis frameworks and tools, as well as analysis workflows and performance metrics embedded in methodologies.

### 2.1 Performance Analysis Frameworks and Tools

This section presents an overview of popular performance analysis frameworks and tools. Performance analysis tools can be broadly categorized into online monitoring and postmortem tools. Monitoring in this context means during execution and postmortem after execution [14].

On the next lower level performance analysis, tools can be distinguished based on their techniques for collecting performance data, namely sampling and instrumentation. Sampling is an interrupt-based technique, which records performance data at specific time intervals. Instrumentation on the other hand, is an event-based technique, that records performance data when specific events are reached in the code. While sampling introduces less overhead, instrumentation makes sure that all events of interest are recorded.

Profiling and tracing are the next lower levels of differentiating. Profiling is a method that aggregates and summarizes event information during runtime, generating a profile. Profiles contain aggregated information about the application. Tracing is a method that collects individual event information and records them as an event flow, generating a trace. Traces contain information about individual events on a timeline. In contrast to profiles, traces contain much more detailed information and can get very large.

Visualization of profiles and traces in the form of graphs and charts is another aspect of performance analysis tools. Not all tools provide visualization, while other tools are dedicated to only provide visualisation.

### 2.1.1 Score-P

Score-P is a scalable performance measurement infrastructure for parallel codes, that supports profiling and tracing of parallel applications [24]. Score-P is an instrumentation and measurement system with connection points to popular performance analysis tools, such as: Scalasca, Vampir, and TAU.

#### Joint Measurement Infrastructure

Many performance analysis tools have overlapping and redundant basic functionality. Examples are instrumentation, profiling, tracing, and data formats. Score-P aims to present a joint approach to offer basic functionality. The Score-P components are: an instrumentation framework, runtime libraries, and helper tools. The instrumentation framework enables the insertion of measurement probes into C, C++, and Fortran code. These probes record performance metrics when they are triggered during execution of the application. It supports the programming paradigms MPI, SHMEM, OpenMP, Pthreads, CUDA, and OpenCL.

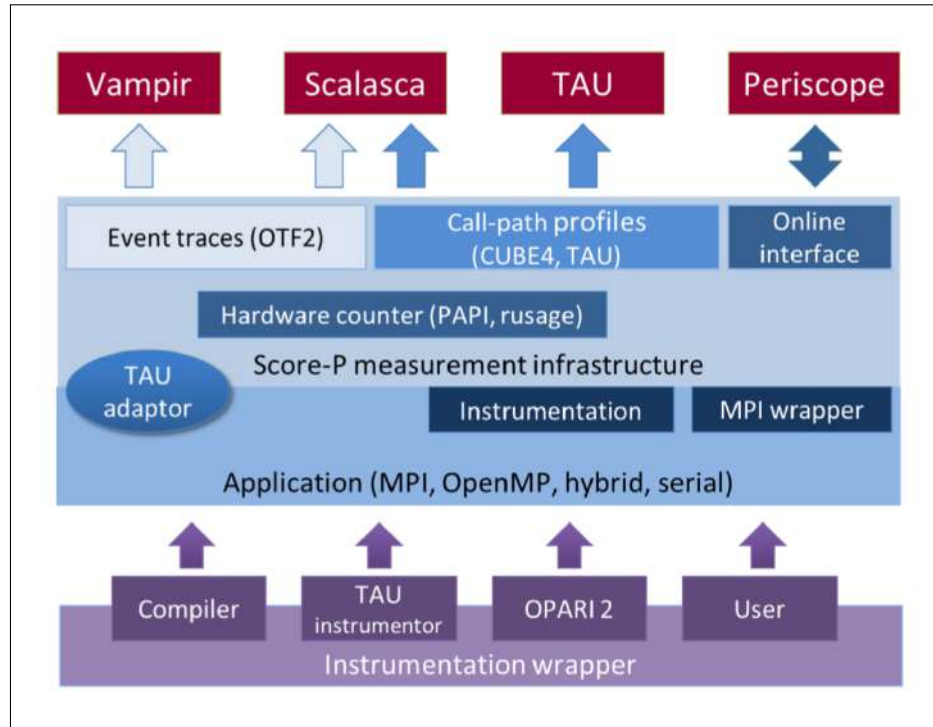


Figure 2.1: The Score-P architecture. It contains the components OTF2 (Open Trace Format Version 2), CUBE4 (a data model and profiling format), and OPARI2 (an instrumenter). It also shows connection points to performance analysis tools such as Periscope, that monitors applications during runtime [10]. Reprinted from [24].

## Architecture

Figure 2.1 shows an overview of the Score-P architecture and also the supported analysis tools [24]. The overview contains the high-level components OTF2, CUBE4, and OPARI2.

- OTF2 (Open Trace Format Version 2) is a software package developed with regards to the two predecessor formats OTF1 and EPILOG, which are the native formats of VampirTrace and Scalasca [12].
- CUBE4 is a profiling format and data model [13]. It represents the behaviour of an application along three dimensions. The first dimension contains performance metrics. The second dimension contains the call tree (location of a certain issue). The third dimension contains the system description.
- OPARI2 is an instrumenter used to wrap OpenMP constructs with calls to the performance monitoring interface POMP [28].

## Summary

Score-P provides a single platform for performance measurements of parallel applications, while supporting different languages (C, C++, and Fortran) and programming models (MPI, SHMEM, OpenMP, Pthreads, CUDA, and OpenCL). The measurement results of Score-P can be explored using several different analysis tools like Scalasca, Vampir, and TAU. The joint measurement framework succeeds in offering uniform access to otherwise redundant basic functionality of individual tools.

### 2.1.2 TAU

TAU (Tuning and Analysis Utilities) is an open-source performance evaluation tool-set developed to support profiling and tracing of parallel applications [33]. TAU offers instrumentation, measurement, and analysis. PDT (Program Database Toolkit) was developed as part of TAU and offers automatic instrumentation. Profile visualization is done via ParaProf [9], which is included in the TAU distribution. Data mining is done via PerfExplorer [19]. Figure 2.2 and 2.3 provide an excellent overview of the TAU architecture, consisting of the high level components instrumentation, measurement, and analysis.

## Instrumentation and Measurement

An overview of the instrumentation and measurement components is given in Figure 2.2. Instrumentation involves source code instrumentation by using pre-processors and compiler scripts, wrapping external libraries (MPI, CUDA, OpenCL), and rewriting the binary executable. Measurement involves hardware counters, profiling, and/or tracing. TAU generates performance data by instrumenting functions, methods, and statements. It is also capable of event-based sampling [33]. TAU uses its own binary trace format but has a built in trace translator, which makes TAU compatible with other tools like Vampir.

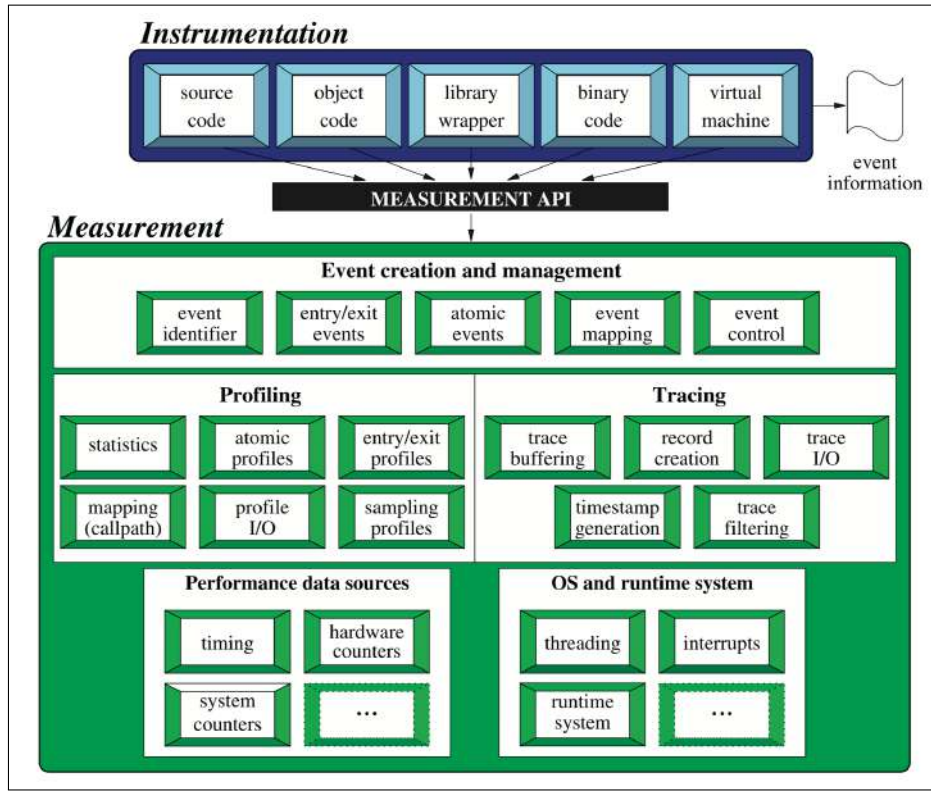


Figure 2.2: Overview of instrumentation and measurement provided by the TAU framework, showing details for Profiling, Tracing, and other components. Reprinted from [33].

## Analysis and Visualization

An overview of the analysis and visualization components is given in Figure 2.3. Analysis involves visualisation of the generated profiles and traces. ParaProf offers visualisation of profiles. TAU is also capable of tracing which opens investigation by third-party trace visualizers like Vampir or Paraver [33].

## Summary

TAU is compatible with a lot of different tools and therefore holds a central position among them. It combines instrumentation, measurement, analysis, and visualization by use of profiling and tracing applications written in various languages (C, C++, Fortran, UPC, Java, Python, and Chapel) and with different programming models (MPI, OpenSHMEM, ARMCI, PGAS, DMAPP, Pthreads, OpenMP, OMPT, GPU, CUDA, OpenCL, and OpenACC).

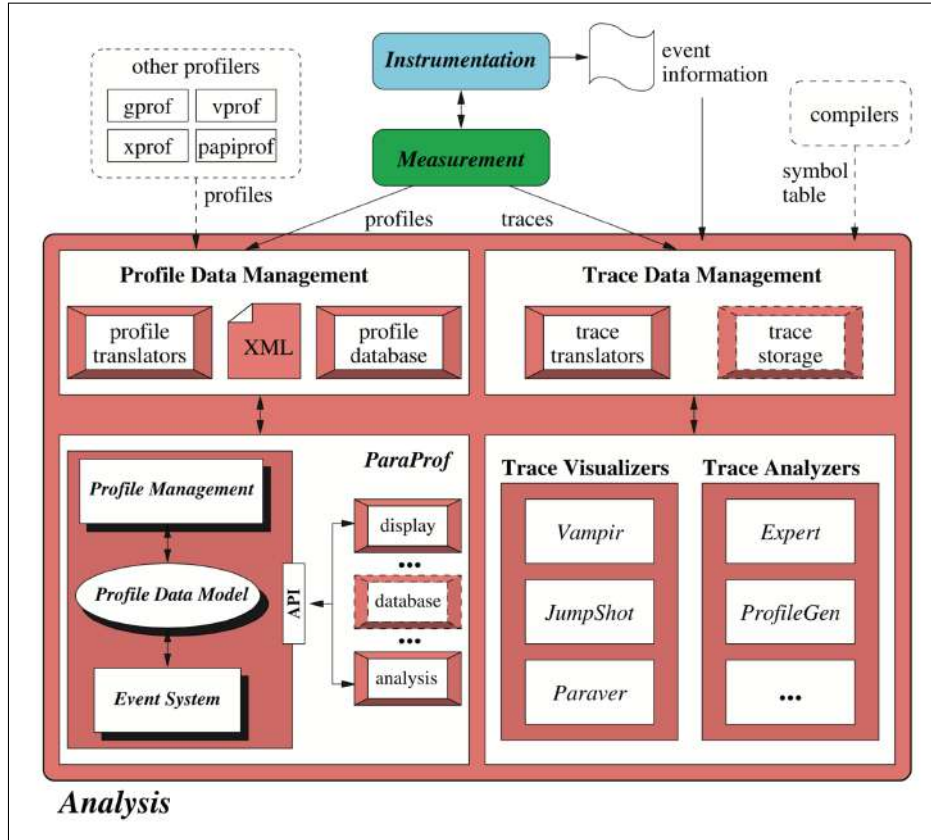


Figure 2.3: Overview of the analysis and visualization provided by the TAU framework, showing details for Profile Data Management, Trace Data Management and corresponding visualisation. Reprinted from [33].

### 2.1.3 Scalasca

Scalasca (Scalable performance Analysis of large-Scale parallel Applications) is a tool-set specifically designed for the analysis of large-scale systems [14]. Scalasca offers an analysis procedure which includes runtime summaries, event tracing, and a unique ability to identify wait states. The basic analysis workflow of Scalasca is shown in Figure 2.4 reprinted from [14], which explores the different phases when working with Scalasca. It starts with the instrumentation of a target application, followed by measurement of representative data, the analysis of said data and the presentation of results in a report.

#### Instrumentation and Measurement

The first step in the Scalasca workflow is to instrument the target application. The user is offered a manual and automatic instrumentation option. By prefixing compilation commands with the Scalasca instrumenter (based on Score-P), the application is linked to the measurement library.



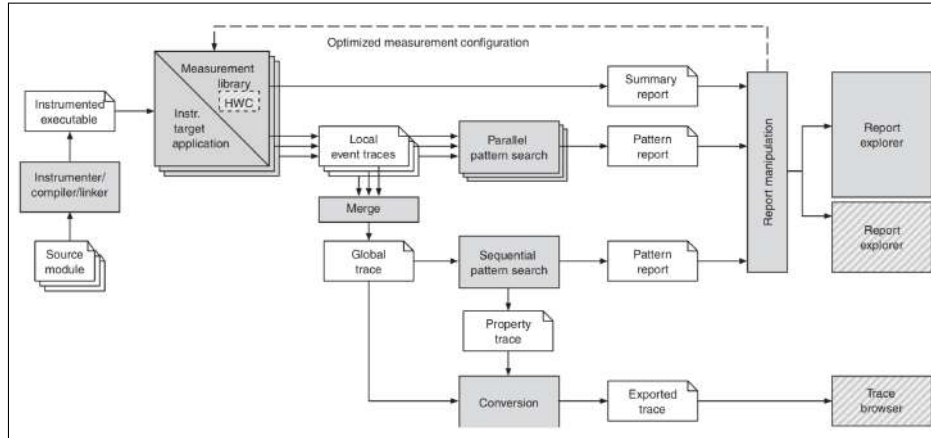


Figure 2.4: Performance data flow of Scalasca. Gray stands for programs, white for files, and hatched boxes represent third-party components. Reprinted from [14].

### Event Summarization and Analysis

After the instrumentation process the user can choose to enable a summary report and/or a pattern-analysis report [14]. The summary report is a runtime summarization of performance metrics and a compact representation of execution behaviour (profiling). The pattern-analysis report uses event traces for the analysis (tracing). The trace files are analysed by Scalasca in parallel by using the same number of computing units as the original application used during execution. The pattern-analysis report contains information about communication and synchronization inefficiencies. Scalasca derives inefficiencies with its ability to detect characteristic patterns that indicate wait states and other performance properties. Alternatively the trace files can also be analysed with the help of a visualization tool like Vampir.

### Report Manipulation and Exploration

Both the summary and pattern-analysis report are XML files, which can be explored with the interactive analysis report explorer of Scalasca [14]. There are sophisticated options for report combination, manipulation (hide specific phases), and comparison (calculate differences).

### Summary

Scalasca is a performance analysis tool-set focused on large-scale systems with many thousand computing units. It offers powerful automatic and parallel trace analysis. Scalasca does not include a trace browser (or visualizer), but is highly compatible with third-party tools like Vampir, TAU and Paraver. Figure 2.5 shows an overview of the architecture of Scalasca [14], including its different components, as well as functional and temporal analysis phases. It also shows where analysis can be extended by suitable third-party programs.

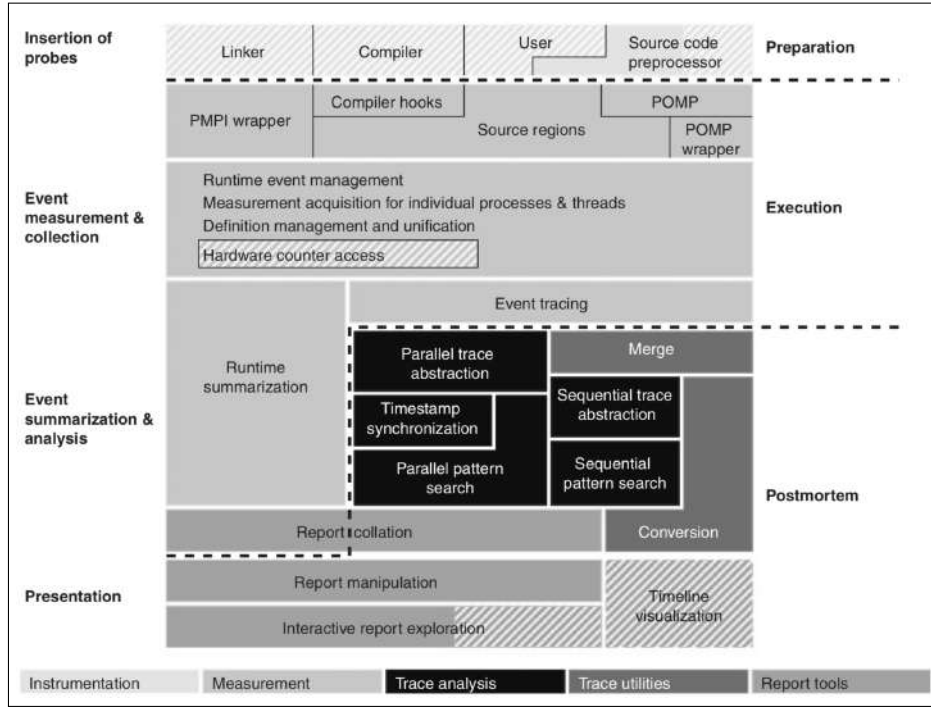


Figure 2.5: The architecture of Scalasca. Each box is a component of Scalasca (third-party components are hatched). Analysis phases are shown vertically and different options in each phase are shown horizontally. Functional phases are described on the left and temporal phases on the right. Reprinted from [14].

#### 2.1.4 Vampir

Vampir (Visualization and Analysis of MPI Resources) is a commercial performance analysis tool. It was originally developed to support the analysis and visualization of MPI applications [29] and later extended to support other programming models such as OpenMP [23]. Vampir is using the OTF2 trace format. Related tools are the open source VampirTrace, as well as the commercial product VampirServer. Despite the existence of VampirTrace, Score-P [24] is currently recommended as code instrumentation and runtime measurement framework for Vampir.

## Vampir Tool Family

The Vampir tool family enables visualization and parallel performance analysis via profiling and tracing. Vampir translates trace files into graphical views which include timeline and statistical charts. VampirTrace performs instrumentation and runtime measurement. Instrumentation can be automatic by using special compiler flags, or manual by specific API calls which mark functions or code regions. Runtime measurement includes the recording of: hardware performance counters, memory usage, I/O activity, etc. VampirServer is the parallel successor of Vampir and implemented in a client-server framework, as shown in Figure 2.6. It introduces parallel analysis which increases scalability [4]. In the following we will focus on the visualization tool Vampir.

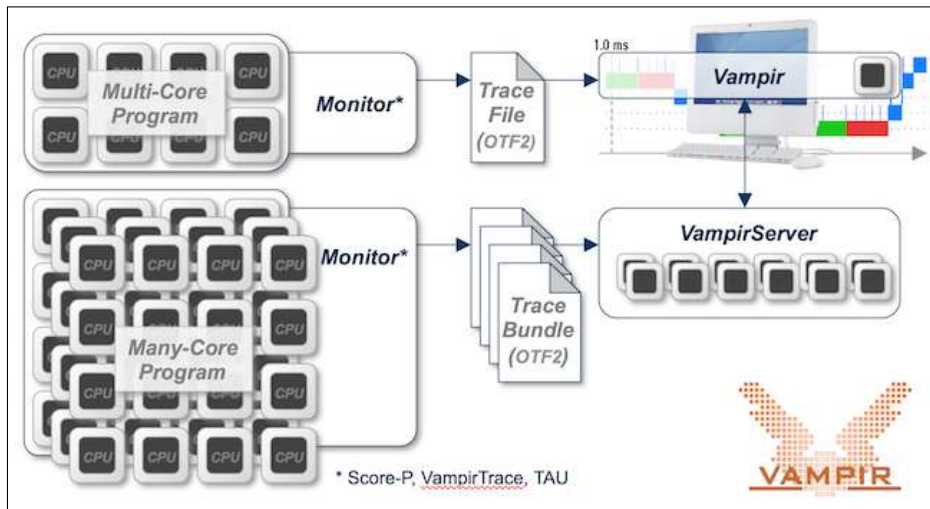


Figure 2.6: Comparison of Vampir and VampirServer setup. Highlighting the increased scalability and client-server framework of VampirServer. Reprinted from [4].

## Timeline Charts

The timeline charts of Vampir show recorded individual event information on a time axis (the chain of events). They enable analysis of the behaviour and event flow of an application. Vampir offers several different types of charts, we show the master timeline in Figure 2.7 which shows information about function, communication and synchronization events for all processes [5].

Functions have different colours according to their function group. For example `MPI.Wait` belongs to the MPI function group. It is possible to distinguish different application phases at a given time, on the basis of the function colours.

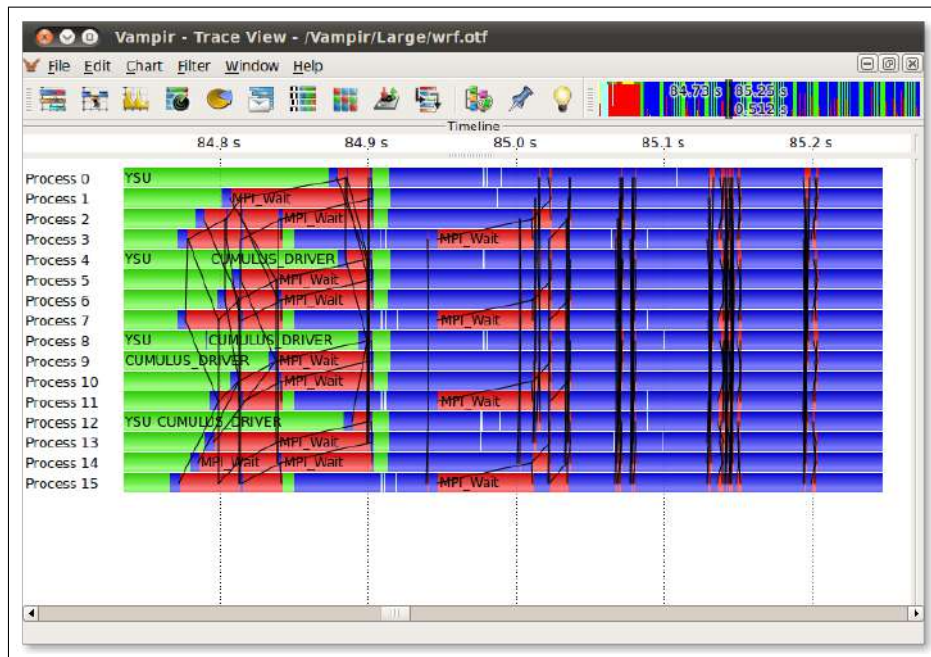


Figure 2.7: The master timeline is the main view and starting point for performance analysis. It shows information about function, communication and synchronization events for all processes. Reprinted from [5].

### Process and Counter Data Timeline

The process timeline holds the same information as the master timeline, only for one individual process and divided into call stack levels of function calls. The counter data timeline shows the value of a counter during the execution time of the application. Counter values can be floating point operations or cache misses.

### Statistical Charts

The statistical charts of Vampir show summarized event information for selected time intervals. Vampir provides different charts highlighting different aspects of the application. The communication matrix is shown in Figure 2.8 which shows information about messages which are sent between processes. The function summary and call tree view are briefly discussed below.

### Function Summary and Call Tree

The function summary shows the aggregated elapsed time of individual functions and function groups. The call tree contains the invocation hierarchy of all functions, as well as the number of invocations and the time spent in the respective calls.

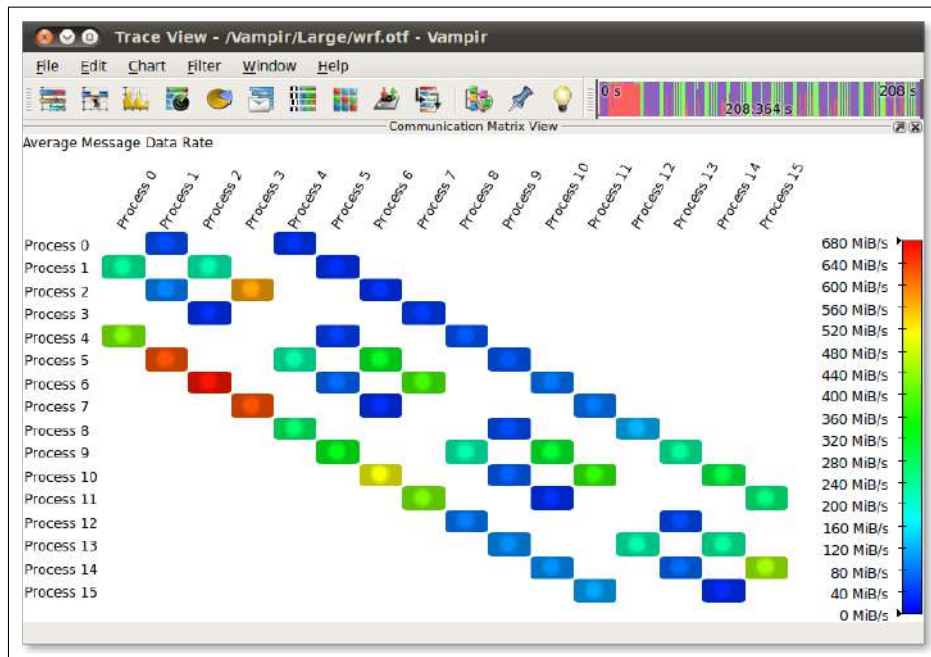


Figure 2.8: The communication matrix shows information about messages which are sent between processes. This view offers different metrics like: number of messages, average message size, average message data rate, etc. Reprinted from [5]

## Summary

Vampir provides powerful features for the performance analysis of parallel applications. The different timeline and statistical charts lift analysis to a user-friendly graphical level and help understand application behaviour and individual event flow.

### 2.1.5 HPCToolkit

HPCToolkit is an open-source tool-set for application performance analysis [6]. HPCToolkit supports profiling and tracing. It achieves a low overhead of 1-5% by using sampling to generate profiles [2] and is therefore able to scale to large parallel systems. HPCToolkit also comes with presentation tools, which allow for visualization of various application characteristics. The tool-set supports different programming models (MPI, OpenMP, Hybrid and Pthreads).

## General Methodology

Figure 2.9 shows the primary components of HPCToolkit and the workflow that combines them. What follows is the basic structure of the workflow, taken from [6]:

1. Measurement of performance metrics while an application executes.
2. Analysis of application binaries to recover the program structure.
3. Correlation of dynamic performance metrics with source code structure.
4. Presentation of performance metrics and associated source code.

## Workflow

The following paragraph explains the workflow depicted in Figure 2.9. First compile and link the target application. Second run the application with `hpcrun`, which generates a profile using sampling. Third use `hpcstruct` to analyse the application binary. Fourth use `hpcprof` which combines performance metrics with the structure of an application into a performance database. Fifth use `hpcviewer` to explore the generated performance database.

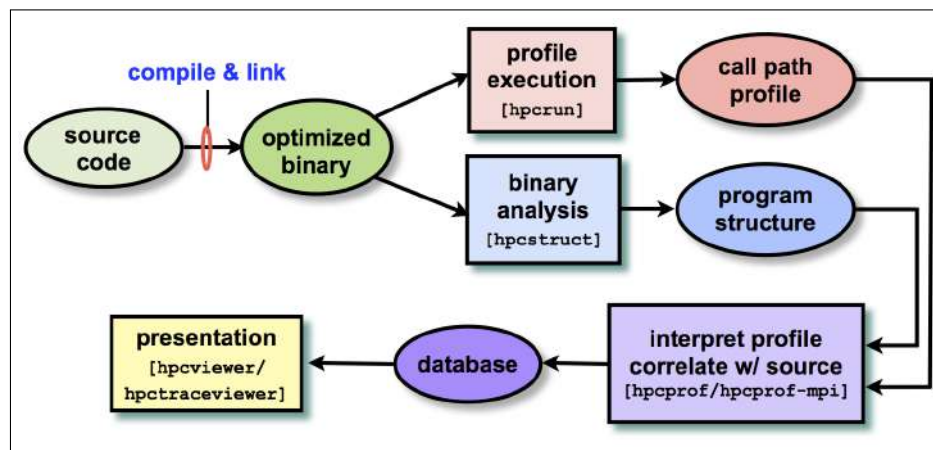


Figure 2.9: Showing the main components of HPCToolkit and the underlying performance analysis workflow that combines them. Reprinted from [2].

### **hpcrun**

Calling-context-sensitive performance measurements are collected by the component `hpcrun` [2]. It is using system timers and performance monitoring unit (PMU) events to trigger sampling, which achieves a low overhead of 1-5%. the `hpcrun` component is collecting call path profiles and also has the option to enable tracing.

### **hpcstruct**

The calling-context-sensitive measurements which are collected by `hpcrun` are associated with the source code structure of the target application by the component `hpcstruct` [2]. It generates information about the relationship between the application binary and its source code. the `hpcstruct` component identifies relations between object code and source code, procedures, and loop nests. It also identifies inlined code.

### **hpcprof**

The `hpcprof` component takes call path profiles and traces generated by `hpcrun`, overlays them with the application structure from `hpcstruct` and correlates the result with source code [2]. The sub-component `hpcprof/mpi` is able to do this correlation in parallel. and can handle thousands of profiles from a parallel execution. The result is a performance database which can be presented with `hpcviewer` and `hpctraceviewer`.

### **hpcviewer**

One of the visualization components is `hpcviewer`, which presents performance data and provides a graphical view of performance variability across processes and threads [2]. The component is designed to highlight scalability losses and inefficiencies instead of only focusing on application hot spots.

### **hpctraceviewer**

The other visualization component is `hpctraceviewer`, which shows the program execution behaviour along a time axis [2]. The `hpctraceviewer` component presents activity over time at different call stack depths and therefore renders traces at multiple levels of abstraction.

### **Summary**

HPCToolkit is a very structured tool-set, where every component has a clear role and distinct function. The provided methodology, which is embodied in the workflow shown in Figure 2.9, provides a good step-by-step guide utilizing the full potential of HPCToolkit and arriving at a qualitative and quantitative analysis including timeline and statistical information.

## 2.1.6 Paraver

Paraver was developed to visualize and analyse parallel applications after execution [31]. A key feature is the capability to handle large trace files by using filters to block uninteresting information and summarize data. In order to filter information. Paraver uses a method called soft counters [14]. Paraver uses its own distinct trace format without semantics, which makes it easy to upgrade the tool for new performance data or programming models. The trace files are generated by the runtime measurement system Extrae [1]. Metrics of Paraver are very flexible because they can be programmed by the user, with the help of various time functions and filter modules. Additionally Paraver has convenient features like multi-trace comparison and cooperative analysis.

### Visualization

Paraver visualizes trace files through a minimal set of views [1]. The timeline display is shown in Figure 2.11 and the statistics display in Figure 2.10. The timeline display is very similar to the master timeline view of Vampir, representing the application behaviour over time. The statistics display shows quantitative analysis data. The user can choose specific values and events which are passed either to the visualization module or the quantitative representation, as well as colours and scales. The many options for adjustments make Paraver a highly flexible tool.

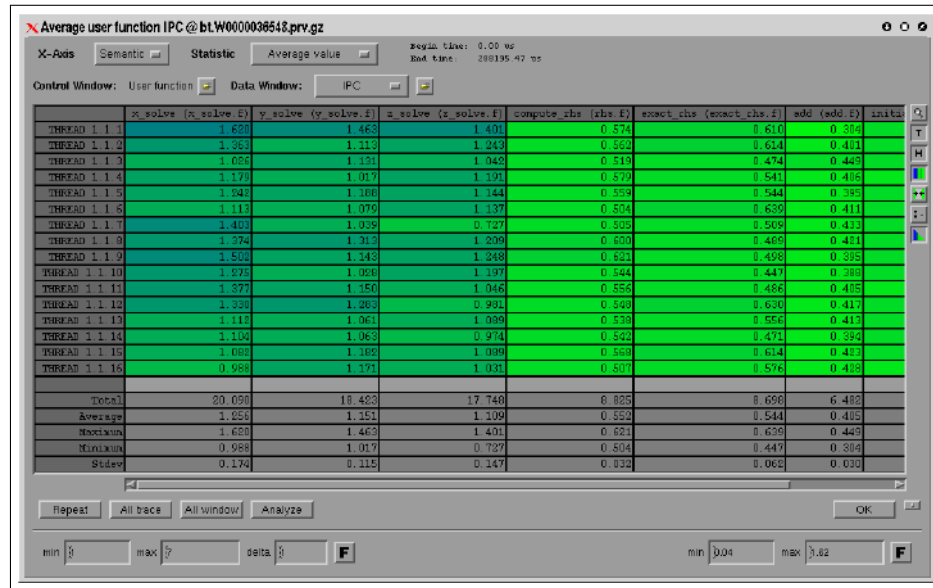


Figure 2.10: The statistics display of Paraver shows summarized numerical analysis of the target application. It is the quantitative analysis part of Paraver. Reprinted from [1].



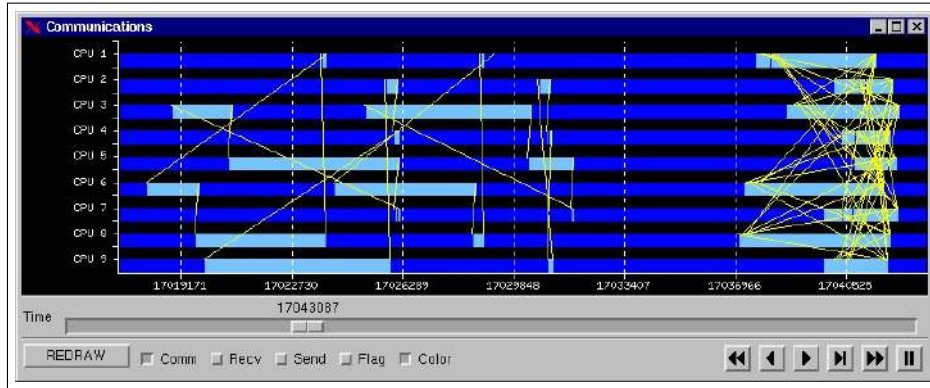


Figure 2.11: The timeline display offers an overview of the application behaviour along a time axis, this allows identification of phases, patterns, and communication events. Reprinted from [1].

### Expressive Power

The semantic module decides about the values which are to be displayed. The user is given access to basic semantic functions (Sum, Sign, Last Event Value, etc.) on every level. These basic functions can be combined to powerful representations of the application. Possible values are: total per CPU consumption when several tasks share a node, evolution of the value of a selected variable, instructions per cycle executed by each thread, etc. [1].

### Summary

Paraver offers a pragmatic approach for application analysis. The tool is capable of handling large trace files and multiple programming models through its distinctive trace format. Paraver offers a high number of possible adjustments that allow the user to combine metrics and chose the way they are visualized. Paraver proved to be highly adaptive and open for new programming models and performance metrics, making it a future oriented tool.

#### 2.1.7 mpiP

The mpiP library provides lightweight and scalable profiling for MPI applications by using the MPI profiling interface layer [36]. It generates less overhead and data than tracing tools, by only accumulating statistical measurements for MPI library routines used by each process. The results collected by mpiP are task-local and in order to merge these results mpiP uses communication at the end of the experiment. Merging the task-local results generates a report which is stored in a textual output file. The mpiP library supports several programming languages (C, C++, Fortran) and was tested with up to 262144 processes [36].

## Sample Output

Sample output from mpiP using a simple application with 4 MPI calls, is shown in the following paragraphs. Starting with Figure 2.12 that shows general information about the execution, mpiP version, etc.. The outputs are reprinted from [36]. It is important to note that mpiP omits certain local MPI calls, such as `MPI_Comm_size`, in order to reduce perturbation. The documentation also mentions the Qt mpiP viewer which provides visualization for the reports [36].

```
@ mpiP
@ Command : /g/g0/chcham/mpiP/devo/testing/./9-test-mpip-time.exe
@ Version : 2.8.2
@ MPIP Build date : Jan 10 2005, 15:15:47
@ Start time : 2005 01 10 16:01:32
@ Stop time : 2005 01 10 16:01:42
@ Timer Used : gettimeofday
@ MPIP env var : -t 10.0
@ Collector Rank : 0
@ Collector PID : 25972
@ Final Output Dir : .
@ MPI Task Assignment : 0 mcr88
@ MPI Task Assignment : 1 mcr88
@ MPI Task Assignment : 2 mcr89
@ MPI Task Assignment : 3 mcr89
```

Figure 2.12: Showing header information provided by mpiP, which contains general information about the execution, mpiP version, etc. Reprinted from [36].

In the following paragraphs we will show an excerpt of the textual output report of mpiP, which gives an insight of the capabilities and information provided by this MPI profiling library. Figure 2.13 shows a list of all tasks used during the execution of the application, and their individual time spent in MPI [36]. Apptime measures the time between `MPI_Init` and `MPI_Finalize`. MPITime measure the time for MPI calls within Apptime. MPI% denotes the ratio of MPITime to Apptime. (\*) is the aggregated time.

@--- MPI Time (seconds) ---@			
Task	AppTime	MPITime	MPI%
0	10	0.000243	0.00
1	10	10	99.92
2	10	10	99.92
3	10	10	99.92
*	40	30	74.94

Figure 2.13: An overview of MPI Time in seconds using the wall-clock time is part of the data collected by mpiP. Reprinted from [36].

Figure 2.14 shows MPI callsites of the application [36]. In order of columns: ID is the callsite ID, Level is the stack depth, File/Address provides the file name, Line denotes the line number, Parent\_Funct is the parent function, and the last column contains the type of MPI call without MPI\_ prefix.

@--- Callsites: 2 -----					
ID	Lev	File/Address	Line	Parent_Funct	MPI_Call
1	0	9-test-mpip-time.c	52	main	Barrier
2	0	9-test-mpip-time.c	61	main	Barrier

Figure 2.14: MPI callsites of the application are also tracked by mpiP, together with Line, Parent Function and the corresponding MPI call. Reprinted from [36].

Figure 2.15 shows an overview of the top callsites regarding time consumption in the target application [36]. In order of columns: Call is the type of MPI function, Site is the callsite ID, Time denotes the aggregate time in milliseconds, App% is the ratio to total application time, MPI% is the ratio to total MPI time, COV holds the coefficient of variation from the individual process time (indicating variation in times of individual processes).

@--- Aggregate Time (top twenty, descending, milliseconds) -----					
Call	Site	Time	App%	MPI%	COV
Barrier	2	3e+04	75.00	100.00	0.67
Barrier	1	0.405	0.00	0.00	0.59

Figure 2.15: Top callsites with regards to aggregated time consumption is also part of the mpiP output. Reprinted from [36].

## Summary

The textual output of mpiP provides statistical information about MPI library routines. Because mpiP is using profiling it generates very low overhead and is easy to setup. On the other hand it does not provide qualitative analysis in the sense of showing timeline behaviour of the application, mpiP enables a quantitative analysis of the application by looking at aggregated performance data.

## 2.1.1.8 Overview and Comparison

Tools	Progr. Languages	Programming Paradigms	Licensing	Sampling / Instrumentation	Profiling / Tracing	Trace Format	Visualization	Performance Analysis
Vampir	C, C++, Fortran	MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC	commercial	-	-	OTF2	yes	postmortem
Scalasca	C, C++, Fortran	MPI, OpenMP, Pthreads	open-source	Instrumentation	Profiling and Tracing	OTF2	no	postmortem
TAU	C, C++, Fortran, UPC, Java, Python, Chapel	MPI, OpenSHMEM, ARMCI, PGAS, DMAPP, Pthreads, OpenMP, OMPT, GPU, CUDA, OpenCL, OpenACC	open-source	Sampling and Instrumentation	Profiling and Tracing	TAU trace format	yes	postmortem
Paraver	C, C++, Fortran, Java, Python	MPI, OpenMP, Pthreads, ompSs, CUDA	open-source	-	-	Paraver trace format	yes	postmortem
HPCToolkit	C, C++, Fortran	MPI, OpenMP, Pthreads	open-source	Sampling	Profiling and Tracing	CCT tuples	yes	postmortem
Score-P	C, C++, Fortran	MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC	open-source	Sampling and Instrumentation	Profiling and Tracing	OTF2	no	-
mpiP	C, C++, Fortran	MPI	open-source	Instrumentation	Profiling	-	no	postmortem

Table 2.1: Comparison of performance analysis tools. CCT (Calling Context Tree) are call paths for events. Score-P and Vampir are often used together. The Paraver trace format is generated by Extrae.

## 2.2 Methodologies and Characteristics

This section contains literature relevant to methodologies, analysis workflows, and performance metrics. We will focus on conceptual approaches and the performance metrics.

### 2.2.1 Oxbow and PADS

Oxbow is a toolkit that supports collecting information about application behaviour [37] [34]. PADS (Performance Analytics Data Store) is a web-based infrastructure that supports collecting, storing, querying, and visualization of information gathered by Oxbow [34]. Oxbow also provides a uniform methodology in the form of a workflow and key metrics. The methodology helps to identify specific application characteristics. A primary goal of Oxbow is to determine how well proxy applications mimic the respective full applications.

#### Oxbow + PADS Workflow

Figure 2.16 shows an overview of the Oxbow and PADS workflow, as explained in [34]. We will now briefly summarize the workflow:

1. Use Oxbow to compile and execute the target application while measuring computation, communication and memory behaviour
2. Store the measured performance data and upload it to the PADS data store as part of the Oxbow workflow.
3. Use PADS to dynamically visualize the collected performance data and compare the behaviour of different applications.

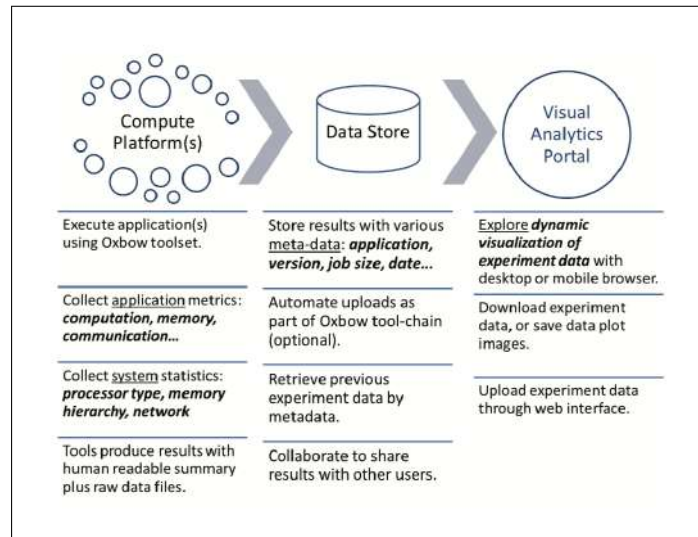


Figure 2.16: Overview of the Oxbow and PADS workflow. Divided into three different phases: data collection, data management, and data visualization. Reprinted from [34]

## Application Characterization

Oxbow differentiate between several different categories of performance metrics. The collected data can be grouped into: computation, communication, memory, and source code. In the following paragraphs we will briefly discuss the aforementioned groups.

### Computation Profiling

The computational profile is defined by Oxbow as the set of executed micro-operations, shown in Table 2.2, reprinted from [34]. These micro-operations are decoded instructions, the decoding is done using MIAMI [27]. MIAMI is a tool-set build on top of Pin [26], a dynamic binary instrumentation tool.

Category	Description
BrOps	Conditional / unconditional branches; direct and indirect jumps.
FpOps	Scalar floating-point arithmetic.
FpSIMD	Vector floating-point arithmetic.
IntOps	Scalar integer arithmetic.
IntSIMD	Vector integer arithmetic.
MemOps	Scalar load and store operations.
MemSIMD	Vector load and store operations.
Moves	Integer and floating-point register copies; data type and precision conversions.
Misc	Other miscellaneous operations, including pop count, memory fence, atomic operations, privileged operations.

Table 2.2: Descriptions of the recognized micro-operations categorized in groups. A high level grouping would be: memory, control, and arithmetic. Reprinted from [34]

### Memory Behaviour Measurement

Oxbow recognizes several metrics for memory behaviour, such as bandwidth. The number of read and writes to compute bandwidth are measured through PAPI [35], which provides access to the performance counter hardware. Oxbow also provides a reuse distance that estimates data locality, as explained in [37]. Reuse distance is defined as how often other data elements are accessed between two consecutive references to an individual data element. This metrics largely indicates cache performance.

## Source Code Analysis

Oxbow is also capable of static analysis of an application's source code. The toolkit can determine language, lines of code, type of parallelism, and number of functions. Oxbow also determines software cyclomatic complexity, which is a metric for the amount of decision logic of an application, further explained in [34].

## Communication Analysis

The analysis is done by investigating the use of the MPI library through mpiP [36], which is a MPI profiling library. Oxbow records point-to-point and collective communication between ranks. Figure 2.17 shows communication matrices of different applications.

The communication matrix gives insight on the message passing behaviour of the target application, and can support finding a communication pattern. For example Figure 2.17(a), the repeated pattern indicates a three-dimensional nearest-neighbour communication, explained in more depth in [37].

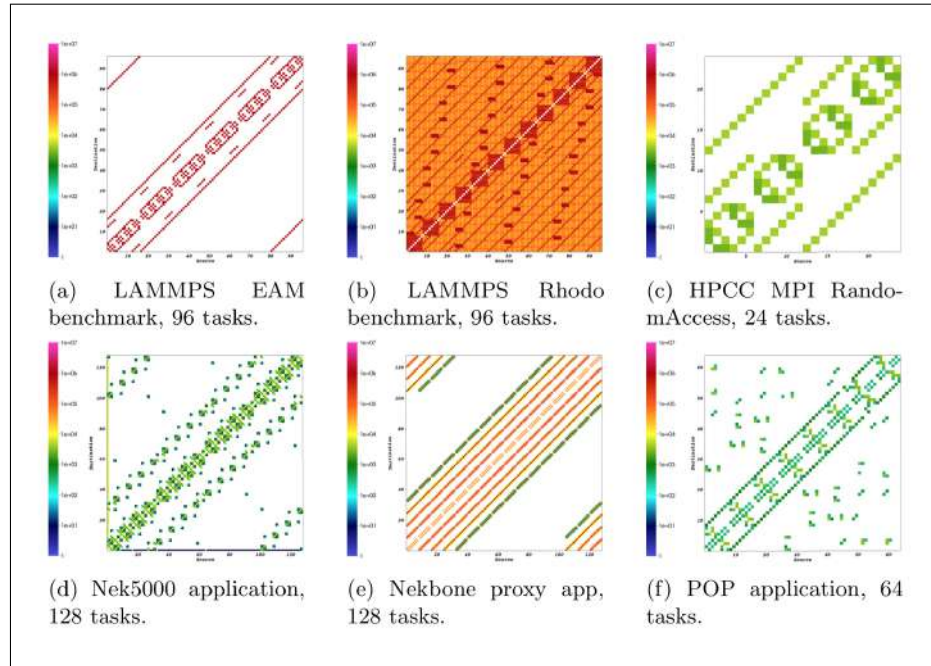


Figure 2.17: Multiple communication matrices for different applications, showing average volume of point-to-point communication and the corresponding communication pattern. Reprinted from [37].

### Trade-offs

In the publications surrounding Oxbow and PADS [37] [34], the authors state a number of trade-offs, which we will briefly summarize in the following paragraph.

- Several input files for MIAMI have to be generated by hand (this problem has been overcome in a newer version according to the authors [34]).
- Different compilers and compilation flags alter the micro-operations recorded by Oxbow. So in order to compare on even ground the same compiler and compilation flags have to be used for all applications.
- The measurement of reuse distance introduces significant overhead, in order to use this feature smaller problem sizes have to be used.

### Clustering of Application Characterization Data

In order to compare performance data of different applications, the toolkit uses a hierarchical clustering with any normalized metric. The result of this clustering process is a dendrogram tree, that groups applications by their similarity, and measures similarity across groups [34].

The clustering is computed by a distance metric that measures the difference between applications. Figure 2.18 shows a clustering with micro-operations as a distance metric, explained in depth in [34]. Micro-operation are defined above as decoded instructions. The tree is built in iterations, each iteration the nearest Neighbors join into a group.

### Summary

The Oxbow toolkit, including PADS, covers almost all aspects of application performance analysis, by relying on several different third-party tools, like mpiP, MIAMI, PAPI, and Pin. The strong point of the toolkit is a well-rounded methodology, which includes several key metrics and a workflow. Including static source code analysis, Oxbow measures different aspects of application performance and PADS supports management and visualization of this performance data.

Interesting aspects are the take on application comparison, using a clustering approach based on similarity, and the consideration of communication patterns. There is no public release at the time of this report, therefore Oxbow is not available for further investigation or comparison beyond the official publications.





### 2.2.2 A Large-Scale Study of MPI Usage

Laguna et. al. [25] provide a study regarding the understanding of state-of-the-practice in MPI usage. The motivation is optimising the communication of HPC applications and identifying the most important MPI features. More than one hundred individual MPI applications were included in the study. They focus on understanding the characteristics of MPI usage with respect to the most used features, code complexity, and programming languages and paradigms.

**Key Characteristics** The study focuses on four key characteristics: (a) the most important MPI routines and features, (b) size and complexity of the applications, (c) MPI usage characteristics versus code release dates, (d) usage of multi-threaded programming models (e.g. MPI+X).

**Scalable Static Application Analysis** Previous work that tried to survey MPI applications relied on profiling the applications, involving instrumentation, compilation and execution. According to Laguna et. al., this approach is difficult to scale due to execution errors and also not portable to all HPC systems. The study therefore employs a scalable analysis method that analyses the code of applications. This approach is considered static, as applications do not need to be compiled and executed. The analysis framework used in the presented study, traverses all directories in the source code and inspects each file to identify the use of MPI calls. They also keep track of programming language, line of code, and whether the application additionally uses a multi-threaded programming model. The analysis framework is written in Python.

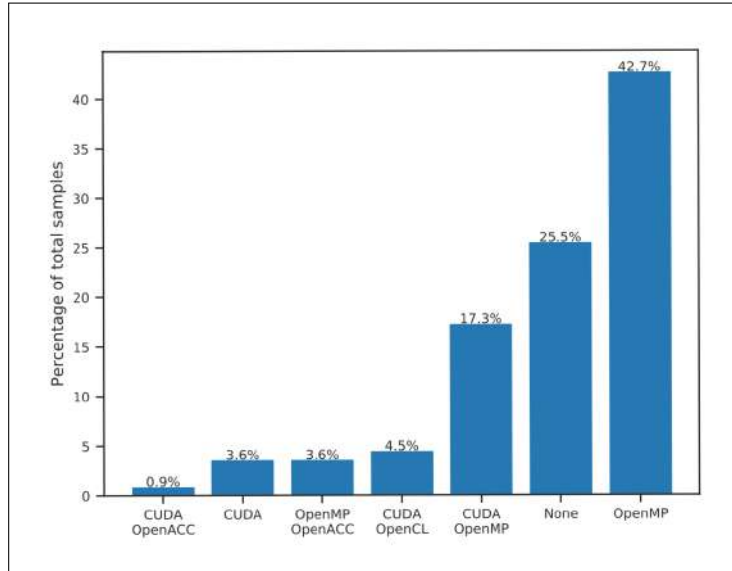


Figure 2.19: Percentages of applications that additionally use a multi-threaded programming model, of these applications, most are using OpenMP. Reprinted from [25].

**MPI Categories** The study presents 13 categories for the routines of the MPI Standard version 3.1. The categories include the main communication types: point-to-point communication, collective communication, and one-sided communication. The categories also include communicator and group management, MPI I/O, and error handling. For a complete list of categories please refer to [25].

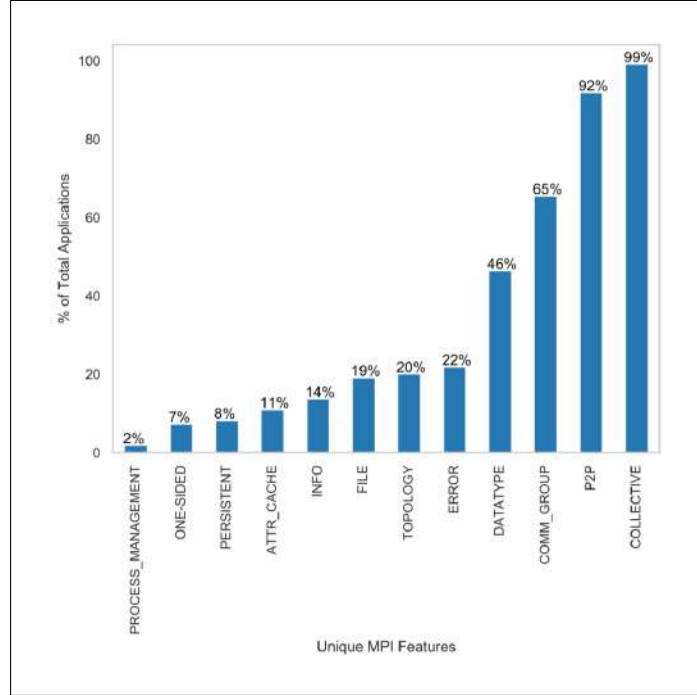


Figure 2.20: Percentage of applications that use specific MPI categories. The most used categories are Point-To-Point and Collective. Reprinted from [25].

**Important Findings** Their most important findings are:

- A large portion of MPI applications do not use advanced features (e.g. persistent or one-sided routines). 67% of applications use blocking send and receive operations.
- The majority of applications use only a small set of MPI features (e.g. a considerable number use only point-to-point and collective communication, leaving other MPI features unused).
- 42% of applications rely on features in MPI version 1.0 only. For about 80% of applications the minimum MPI version they require is 2.0. Features provided by subversions (e.g. 1.3, 2.1, etc.) are rarely used.
- About 2/3 of MPI applications are used together with multi-threaded programming models (OpenMP is the most popular).
- C++ is the dominant language in MPI applications.

### 2.2.3 Benchmark Similarity

Joshi et. al. [22] propose a methodology for measuring the similarity between applications based on their inherent characteristics. The goal of the presented work is to identify a representative subset of programs for benchmark suites and to investigate their evolution over multiple version. The authors work with the benchmark suites: SPEC CPU2000, MediaBench, and MiBench. In the following paragraphs we will summarize the methodology and the clustering of similar applications proposed in the aforementioned work.

#### Characteristics

The authors rely on microarchitecture-independent characteristics to investigate inherent properties of programs. The metrics are explained in detail in [22], and briefly summarized below:

- **Instruction Mix:** Measuring computation instructions, data memory access, and branch instructions.
- **Control Flow Behaviour:** Measuring basic code block sizes and branch directions in the instruction stream.
- **Instruction Level Parallelism (ILP):** Measuring register dependency distance, the number of instructions in the instruction stream between write and read of a register instance.
- **Data Locality:** Measuring average memory reuse distance, the average number of data memory accesses between two consecutive accesses to the same address.
- **Instruction Locality:** Measuring the average number of instructions between two consecutive accesses to the same static instruction.

The characteristics listed above are measured using the custom-grown analyser SCOPE a modification of SimpleScalar [7].

#### Statistical Data Analysis

In order to compare multiple programs with multiple characteristics, the authors use multivariate statistical data analysis. Principal Component Analysis (PCA) [21] is used to remove correlation between the metrics and to reduce the dimensionality of the data set [22]. Cluster Analysis, via k-means and hierarchical clustering [20], is used to find similar groups among the programs. For PCA and hierarchical clustering the presented work uses STATISTICA [22], and for k-means SimPoint [16].

## Similarity Clustering

Joshi et. al. [22] use their aforementioned characteristics and statistical data analysis methods to compare different generations of the SPEC benchmark suite. They look at four SPEC CPU benchmark suites released in: 1989, 1992, 1995, and 2000.

## K-Means Clustering

Figure 2.21 shows the clustering for the four SPEC CPU benchmark suites, using the k-means method [22]. The programs in bold are closest to the center of the cluster.

Cluster 1	gcc(95), gcc(2000)
Cluster 2	mcf(2000)
Cluster 3	<b>turbo3d(95)</b> , applu(95), apsi(95), swim(2000), mgrid(95), wupwise(2000)
Cluster 4	<b>hydro2d(95)</b> , hydro2d(92), wave5(92), su2cor(92), succor(95), apsi(2000), tomcatv(89), tomcatv(92), crafty(2000), art(2000), equake(2000), mdljdp2(92)
Cluster 5	<b>perl(95)</b> , li(89), li(95), compress(92), tomcatv(95), matrix300(89)
Cluster 6	<b>nasa7(92)</b> , nasa(89), swim(95), swim(92), galgel(2000), wave5(95), alvinn(92)
Cluster 7	applu(2000), mgrid(2000)
Cluster 8	<b>doduc(92)</b> , doduc(89), ora(92)
Cluster 9	mdljsp2(92), lucas(2000)
Cluster 10	<b>parser(2000)</b> , twolf(2000), espresso(89), espresso(92), compress(95), go(95), ijpeg(95), vortex(2000)
Cluster 11	<b>fppp(95)</b> , fppp(92), eon(2000), vpr(2000), fppp(89), fma3d(2000), mesa(2000), ammp(2000)
Cluster 12	bzip2(2000), gzip(2000)

Figure 2.21: Clustering for different versions of the SPEC CPU benchmark suite, using the k-means clustering method. Bold written programs are the ones closest to the center of the cluster. Reprinted from [22].

## Hierarchical Clustering

Figure 2.22 shows a dendrogram (tree), which indicates the similarity of programs based on the instruction locality characteristics [22]. The linkage distance generated by the hierarchical clustering corresponds to the vertical scale. Programs are more similar when the linkage distance is shorter.

## Summary

Joshi et. al. [22] present an interesting list of characteristics, that are independent of features of a particular machine configuration. The hierarchical and k-means clustering approaches provide a good method to measure the similarity between different applications.

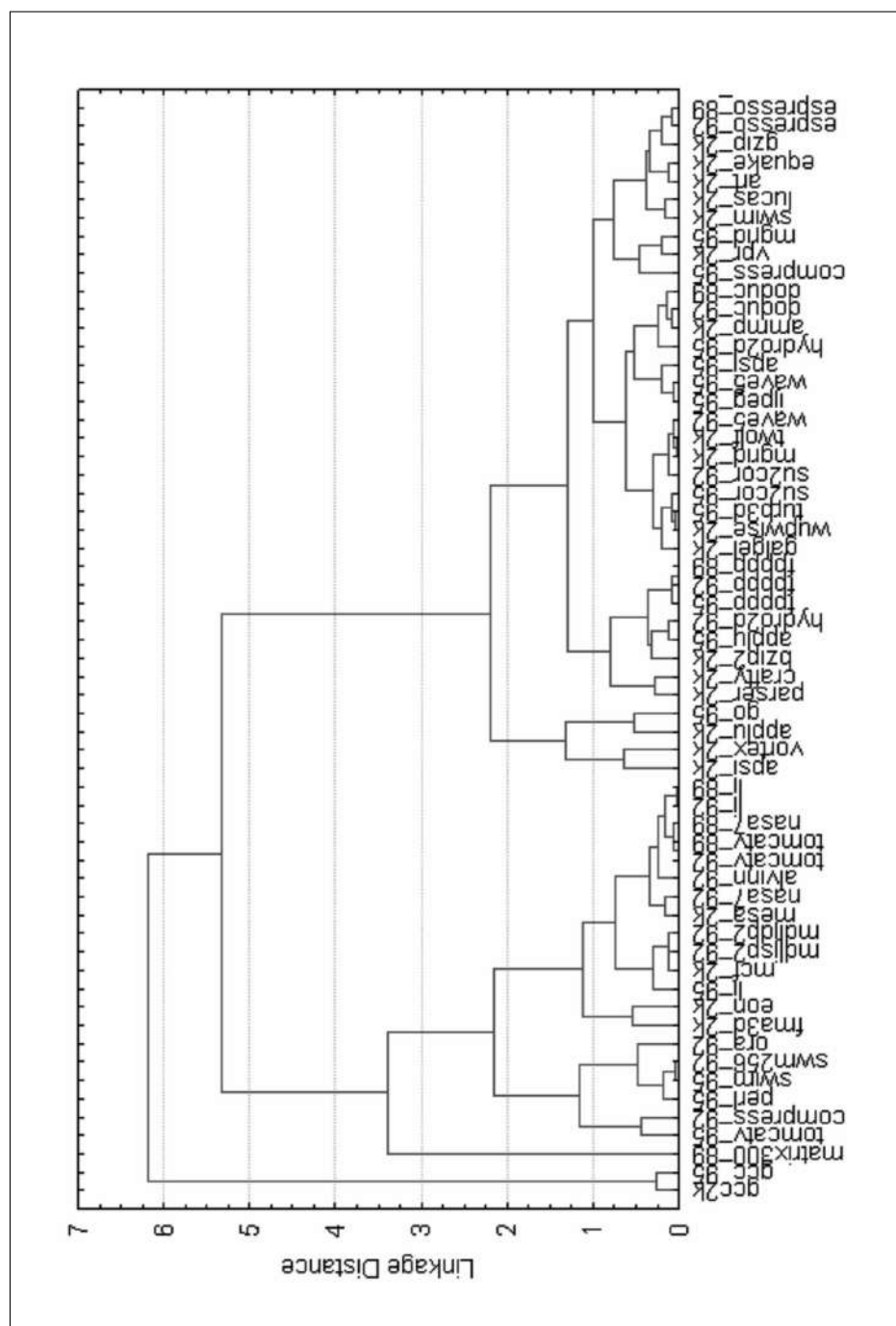


Figure 2.22: Clustering for different versions of the SPEC CPU benchmark suite, using the hierarchical clustering method. Vertical scale depicts the linkage distance. Reprinted from [22].

### 2.2.4 Empirical Performance Evaluation

Vetter et. al. [38] analyse scalability, architectural requirements, and performance characteristics of parallel applications, by following an empirical approach. The aim is to provide a comparative analysis of several different applications focusing on their MPI behaviour. The methods encompass a combination of message tracing and measuring hardware counters. The methodology presented in this work can be summarized as follows: iterative investigation over increasingly refined empirical performance data regarding computation and communication.

#### Measuring Computation Performance

The presented work uses subroutine profiling in order to determine which subroutines take large amounts of wall-clock time. Presenting a hierarchical representation of execution time. Additionally hardware counters are recorded to gain information about processor instructions: number of cycles, number of completed instructions, number of floating-point operations, cache misses, and number of memory loads and stores [38]. Derived from these counters are the metrics: instructions per cycle, computational intensity, and cache hit ratios.

#### Measuring Communication Performance

The profiler mpiP [36] is used to profile MPI communication behaviour in the presented work. Also recorded is the call site stacktrace, which is used to identify different phases of the application [38]. A chronological event stream, for individual calls to the MPI library, is recorded via MPI tracing. This helps investigating load imbalance and the relationship between computation and communication.

#### Communication Pattern

Figure 2.24 shows a communication matrix using Vampir [29], the metric is average message length. The communication matrix shows typical message-passing patterns which are identified by the authors as communication pattern for an iterative linear solver [38]. Communication happens with a predetermined set of processes, additionally the size of the messages increases as the distance between processes decreases.

## Scalability

Figure 2.23 shows the scaling behaviour of two applications, regarding average computation and communication time. One of the shown applications has weak scaling and does not easily scale with the number of processors, while the remaining application shows strong scaling [38].

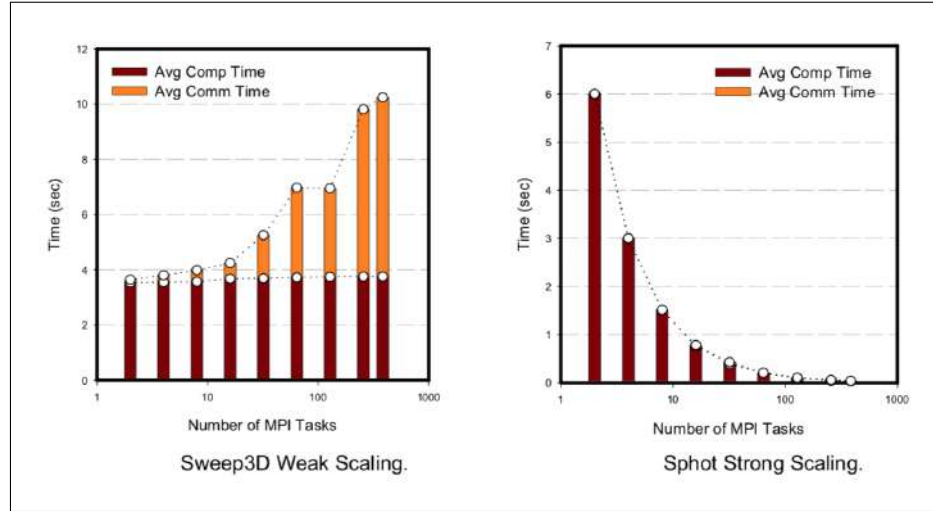


Figure 2.23: Scaling behaviour of several application, regarding average computation and communication time. Reprinted from [38].

## Trade-offs

Many of the application, which are analysed in the presented work, also make use of OpenMP, which is not included in the analysis of the presented work. Other problems encountered by the authors were introduced by tracing. First tracing tends to generate large amounts of data, second tracing can introduce significant perturbation into the target application. The authors limit the problems introduced by tracing through their iterative approach. They first identify potentially interesting code areas without tracing, and in a second step use tracing restricted to these code areas. They were also limited by the amount of information they could gather (through hardware counters) about the applications's memory access pattern.



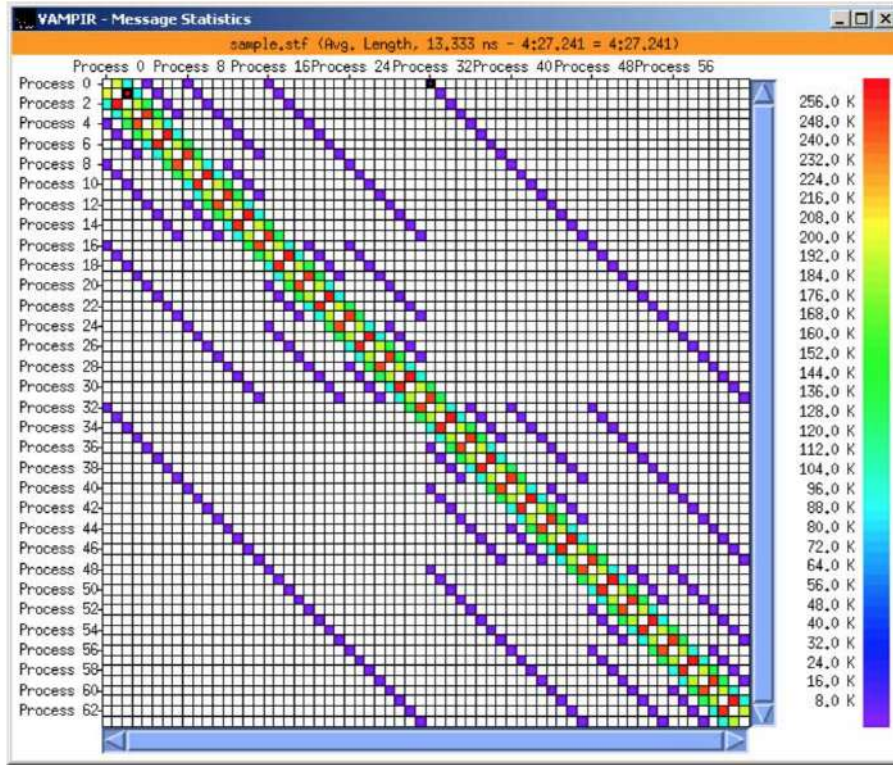


Figure 2.24: Communication matrix generated with Vampir [29], showing message statistics about the average message length between processes, and the communication pattern of the application. Reprinted from [38].

## Summary

The presented work offers an empirical analyses approach to computation and communication analysis of parallel applications. Although the work is focusing on MPI, omitting other programming models, it provides a well-rounded analysis. Including scalability with regards to the relationship of computation and communication, and the identification of typical communication patterns.

### 2.2.5 Communication Patterns

Riesen et. al. [32] investigated metrics that are interesting for the analysis of communication patterns. In the presented work the NAS parallel benchmarks are used as an example. The tool the authors are using is a prototype of a network simulator, which provides recording of message traffic. The network simulator enables the recording message size, message length, as well as source and destination. It can also distinguish between point-to-point and collective communication.

## The Network Simulator

Figure 2.25 shows the experimental setup used in the presented work. When the application uses the MPI library, it also sends an event to the network simulator, the authors achieve this with the MPI profiling interface. The application then waits for an answer of the network simulator before it finishes the MPI call.

The network simulator keeps track of a virtual time which excludes the time spent by the application waiting on an answer of the network simulator. Using this approach with virtual time means, the benchmarks report the same execution time, with or without the network simulator [32].

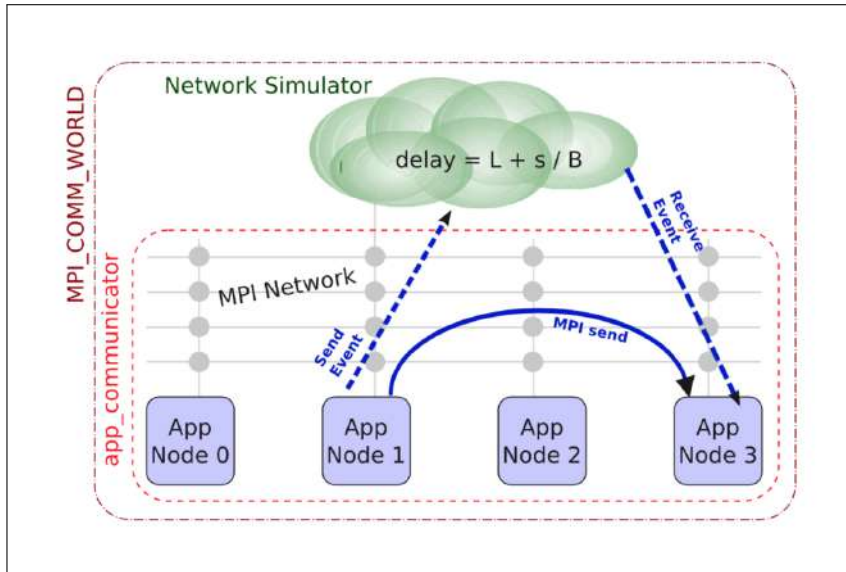


Figure 2.25: Overview of the network simulator.  $L$  is latency,  $B$  is bandwidth, and  $s$  is message size. Node 3 will wait for an event by the network simulator before finishing the MPI send. Reprinted from [32].

## Communication Measurements

Riesen et. al. [32] provide five measurement that help analyse the communication behaviour of a target application. The measurements are presented and briefly summarized in the following paragraphs.

### Message Density

Figure 2.26 and 2.27 shows the message density distribution (communication matrix) for NAS MG. Riesen et.al. [32] concluded that most messages are sent between nodes that are close, regarding MPI's logical node numbering. The result is that nodes build clusters that communicate internal, as nearest Neighbors In the case of 64 nodes there are smaller clusters grouped themselves within bigger clusters. From the diagonal alignment of the clusters, the authors conclude that clusters don't exclusively communicate with nearest Neighbors, but also communicate with the next lower cluster.

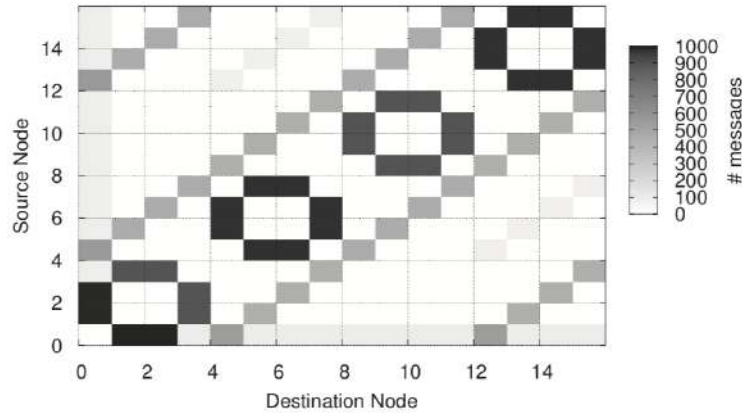


Figure 2.26: Showing message density of NAS MG for 16 nodes. The darker a rectangle, the more messages were sent. Reprinted from [32].

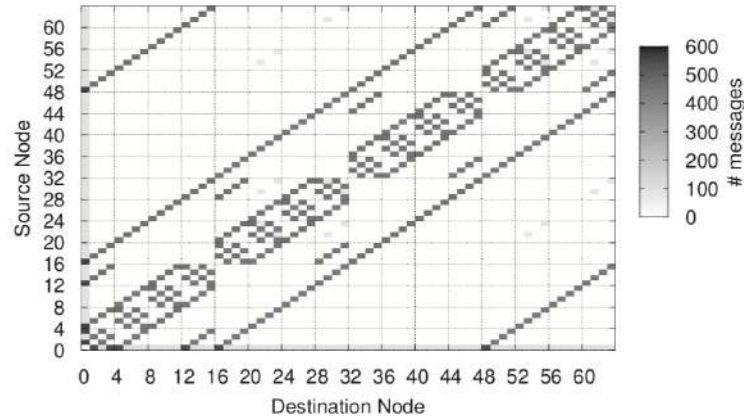


Figure 2.27: Showing message density of NAS MG for 64 nodes. The darker a rectangle, the more messages were sent. Reprinted from [32].

### Data Density

Figure 2.28 and 2.29 shows the message density and data density distribution (communication matrix) for NAS BT. The authors note that some nodes sent more data than they receive, the number of messages can still be the same but the data flow is different [32].

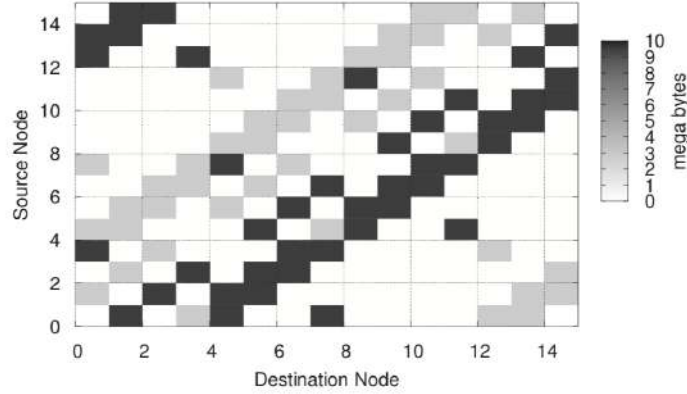


Figure 2.28: Message density of NAS BT. The darker a rectangle, the more messages were sent. Reprinted from [32].

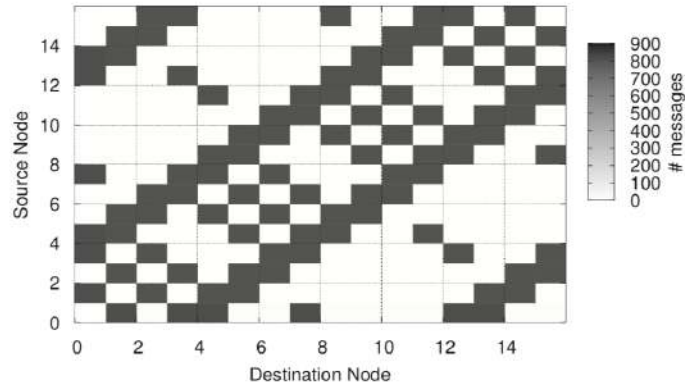


Figure 2.29: Data density of NAS BT. The darker a rectangle, the bigger messages were sent. Reprinted from [32].

### Collectives and Point-to-point

Figure 2.30 shows the ratio of point-to-point to collective communication for several application of NAS [32]. The graphic helps to distinguish between application that rely heavily on one form of communication and application that use both models.

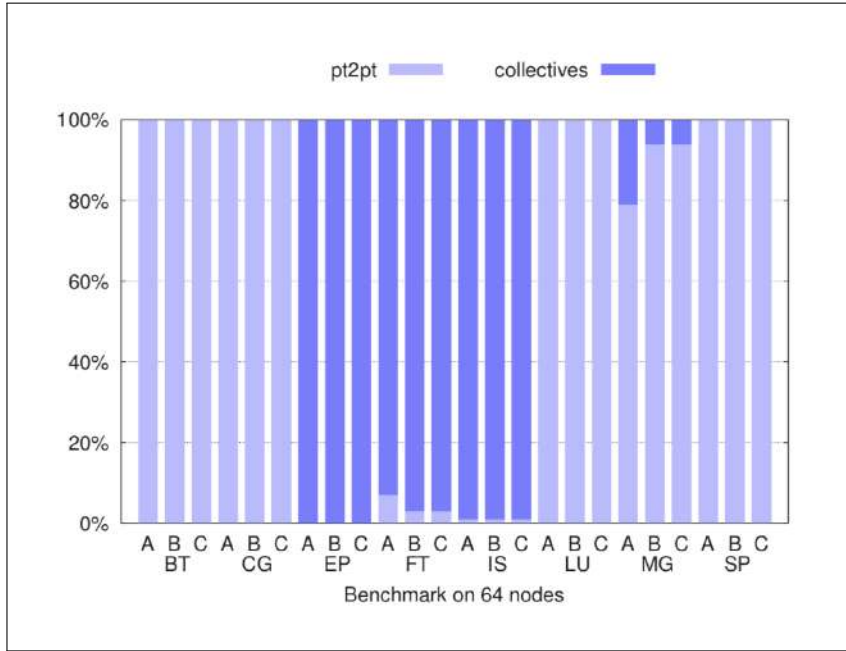


Figure 2.30: Ratio of point-to-point to collective communication for several applications of NAS. The letters A, B, and C indicate the problem size for the individual application. Reprinted from [32].

### Number and Type of Collectives

Figure 2.31 shows which type of collective communication is used by the FT application of NAS. When increasing the number of nodes, the number of MPI\_Bcast (broadcast) also increases, while the other types remain constant. The authors conclude that the other types are only used at the beginning (distribute data) and at the end of the application (collect results), while the broadcast operation scales with the problem size.

### Message Size Distribution

Figure 2.32 shows the message sizes of SP from NAS for an increasing number of nodes. The authors conclude that for SP, the message sizes get smaller when more nodes are used with the same problem size. This is the behaviour of most application of NAS, with some exceptions which are reported by Riesen et. al. in the presented work [32].

### Summary

Riesen et. al. [32] presented a well rounded set of metrics to analyse communication patterns among different applications. The authors also provided insight on how to interpret the metrics and introduced a novel approach to collect communication characteristics. Combining a network simulator with the MPI profiling interface introduces minimal perturbation in the execution of the target application.

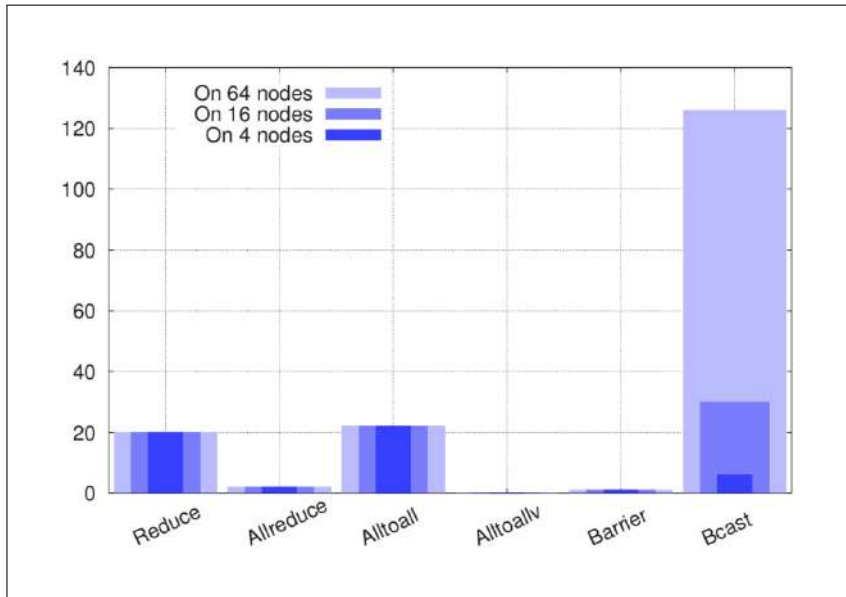


Figure 2.31: Number and type of collective communication used by FT from the benchmark suite NAS, for increasing node counts. Reprinted from [32].

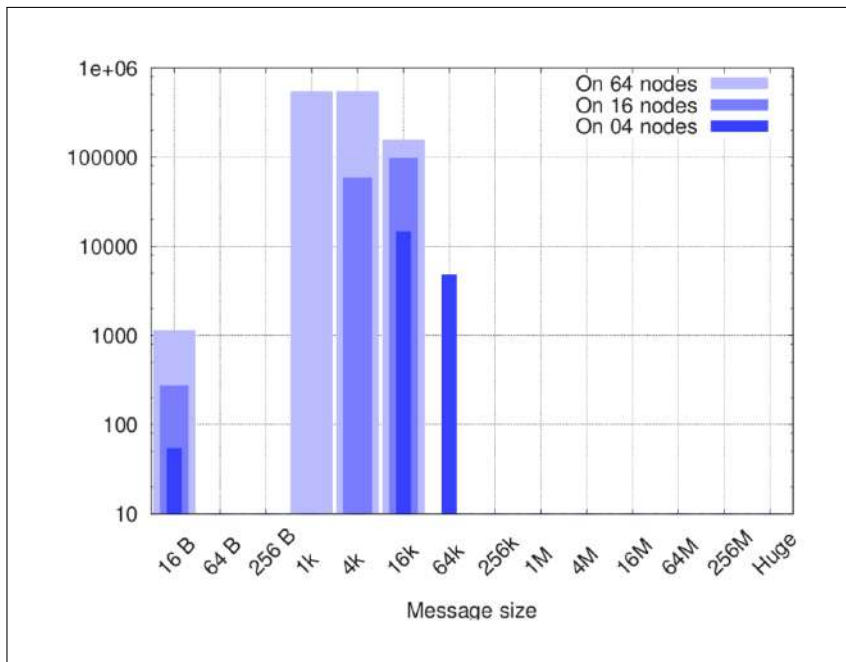


Figure 2.32: Message size distribution for SP from NAS for an increasing number of nodes (using the same problem size for each run). Reprinted from [32].

## Chapter 3

# Proposed Methodology

This chapter introduces the approaches and technologies we used to meet the requirements of our goals and build the Performance Analysis Portal for HPC Applications (PAP).

In the following subsection we will introduce client and server and lay out the plan we made to develop the portal. We will briefly introduce each technology, its usage, and the reasoning for choosing the respective technology.

### 3.1 Server Side - Data Access Layer

The server side, or data access layer, is responsible for page requests and database access. We use a combination of the JavaScript runtime environment Node.js and the database program MongoDB.

#### 3.1.1 Node.js - JavaScript Runtime Environment

Node.js is an open-source runtime environment that organises web development around the programming language JavaScript. It provides a collection of modules that handle different functionality (file system I/O, networking with HTTP and HTTPS, etc.) in order to simplify the creation of web servers.

Node.js is used to build the server side of the portal, serving page requests and handling the database access. This runtime environment was chosen to keep the portal around the programming language JavaScript, and profit from the collection of modules that take care of various core functionalities. We are working with the v10.16.3-linux distribution.

### 3.1.2 MongoDB - Document Oriented NoSQL Database

The original idea was to design an application catalogue around the JSON format without the use of a database program. JSON gives us several advantages:

- Fields can vary from document to document.
- The data structure can be changed over time.
- JSON maps to the JavaScript object which makes it easier to work with.
- JSON is human-readable and can be used as export format

MongoDB is a document-oriented database program. It is in the category of NoSQL databases and uses JSON-like documents as storage method. We decided to use MongoDB as our database program in order to profit from already implemented functionality instead of reinventing the wheel, and to keep in the spirit of using JSON as our format for data storage. We are working with the linux-x86\_64-4.0.12 distribution of MongoDB.

### 3.1.3 PHP - Server Management

PHP was designed for web development and is a general-purpose programming language. PHP can be embedded into HTML code and serve as a pseudo command line access.

We made use of this characteristic of PHP in order to overcome our missing command line access on the production server. We used this setup to install Node.js and MongoDB on the server, run git commands for the repository, and start / stop the Node.js and MongoDB services.

## 3.2 Client Side - Presentation Layer

The presentation layer, or client side, is what the user actually sees from the portal. It presents the functionality of PAP through a graphical user interface which was developed with HTML, CSS, and JavaScript.

### 3.2.1 HTML - Hypertext Markup Language

The standard markup language for web browser content is the Hypertext Markup Language (HTML). It is most often used together with Cascading Style Sheets (CSS) and scripting language JavaScript.

HTML documents are sent from the web server to the web browser in order to be rendered as multimedia pages, they include and semantically describe the structure and appearance of the respective content.



### 3.2.2 CSS - Cascading Style Sheets

Cascading Style Sheets (CSS) is a style sheet language that is used to assist the presentation of HTML documents. The function of CSS is the separation of the content itself, and the descriptions of its intended appearance.

The layout, colours, and fonts can be described outside the HTML document in a separate file, this can reduce complexity and improve readability of the original HTML file. We use a central .css file, that gives a uniform appearance to all web pages provided by PAP.

### 3.2.3 JavaScript - Scripting Language

Usually, JavaScript is written into an HTML page, and used as a client side scripting language. The script is sent to browser together with the HTML page that was requested by the user.

As scripting language it can make web pages interactive. It supports object-oriented and prototype-based programming styles with APIs for text, arrays, dates, regular expressions, and the DOM (Document Object Model) that defines the structure of a document.

### 3.2.4 Google HTML/CSS Style Guide

In order to organize the client side, we decided to follow the Google Style Guide for HTML and CSS <sup>1</sup>. The style guide defines simple formatting rules for HTML and CSS, in order to improve code quality. Some of the basic guidelines are stated below:

- Use UTF-8 as character encoding. Specify the encoding in HTML documents with: `<meta charset="utf-8">`
- HTML5 syntax is preferred for all HTML documents, and specified with: `<!DOCTYPE html>`.
- Use spaces instead of tabs to indent text, because some editors interpret tabs differently.
- Indent by 2 spaces per indentation level.
- Use lowercase for elements and attributes.
- Don't leave trailing spaces at the end of a line.

---

<sup>1</sup>the Google HTML/CSS Style Guide can be found under <https://google.github.io/styleguide/htmlcssguide.html>

### 3.3 Additional Third-Party Libraries

We use multiple third-party JavaScript libraries in order to profit from existing functionality. In the following subsections we will briefly introduce these libraries.

All libraries are downloaded and part of the project, we do not rely on CDN (Content Delivery Network) to dynamically provide the code to the portal. This makes the portal more flexible, because it is not dependent on an internet connection.

#### 3.3.1 jQuery - JavaScript Library

jQuery is a open-source JavaScript library that was developed to support HTML DOM (Document Object Model) manipulation and event handling by making it easier to navigate a document, and select DOM elements. We are using jQuery 3.4.1 <sup>2</sup>.

#### 3.3.2 Plotly.js - Graphing Library

Plotly.js an open-source graphing library that was built on top of d3.js and stack.gl. Plotly.js comes with more than 40 different charts and inbuilt functionality to download charts as .png files. We make use of Plotly.js v1.50.1 <sup>3</sup>.

#### 3.3.3 Mask.js - Input Masking

Mask.js is a jQuery plugin that allows the creation of regular expression masks for input form fields of HTML documents. It helps to keep user input clean and data formats valid. We are using the version v1.14.16 of Mask.js <sup>4</sup>.

#### 3.3.4 Simple Statistics - Statistical Methods

Simple Statistics is a JavaScript library for statistical methods. It supports developers by offering documented and tested statistical functions, ready to be used in JavaScript code. We are using version 7.0.8 <sup>5</sup>.

---

<sup>2</sup>jQuery has been acquired from <https://jquery.com>

<sup>3</sup>Plotly.js has been downloaded from <https://plot.ly/javascript/>

<sup>4</sup>The source for this library is <https://igorescobar.github.io/jQuery-Mask-Plugin/>

<sup>5</sup>has been acquired from <https://simplestatistics.org>

## Chapter 4

# Design and Development

This chapter focuses on design decisions and how we used the technologies presented in the previous chapter to develop the functionality provided by PAP. For descriptions and examples on how the individual functionality is to be used, please refer to Chapter 5 Analysis Methodology.

The portal can be divided into the high level components client and server, the client can further be divided into individual analysis workflow steps. Together this gives us our 4 core "development" parts: (a) server, (b) preparation and measurement, (c) database access, and (d) analysis and visualisation.

Our approach is based on iterative prototyping of different core parts of the portal. After finishing one core part, all the other parts get revisited in order to keep the functionality coherent and avoid introducing bugs.

### 4.1 Server Side - Decisions and Development

Our server was developed to provide the basic functionality of handling page requests and database interaction.

#### 4.1.1 Database Interaction

The interaction with the database is split into three major functions: (a) adding or updating database entries, (b) sending database entries to the client, and (c) deleting database entries.

#### 4.1.2 Computation on the Client

The code and functionality of the server side is intentionally kept small. In order to avoid unnecessary communication between client and server, we developed a lot of the functionality on the client side. This also offers more experienced users a look at the internal logic of the portal by inspecting the JavaScript code, and therefore a better understanding of certain portal behaviour and analysis results.

### 4.1.3 PHP - Command Line Access

To gain command line access to the server, we used PHP embedded into HTML. Using this approach we could open the file shown in Figure 4.1 with a browser and send command through this interface to the command line of the server.

```
1  <?PHP if (isset($_POST['command'])) {  
2      shell_exec($_POST['command'] . ' 2>&1');  
3  } ?>  
4  
5  <title>Pseudo CLI</title>  
6  <h1>Pseudo CLI</h1>  
7  
8  <form action="cli.php" autocomplete="off" method="post">  
9      <p>Command:</p>  
10     <p><input type="text" name="command" value="" size="50" /></p>  
11     <p><input type="submit" style="width: 200px;height: 50px;border: none"  
12         name="send" value="Execute" /></p>  
13 </form>  
14  
15 <input type="button" style="width: 200px;height: 50px;border: none"  
16     value="Back" onclick="window.location.href='https://  
17     performance-analysis.dmi.unibas.ch/manage/manage.html'" />
```

Figure 4.1: Command line access through PHP embedded in HTML code.

### 4.1.4 Dealing with JavaScript Code Injection

In order to handle JavaScript code injection, the server is parsing the uploaded data with `JSON.parse()` instead of `eval()`. `JSON.parse()` is using a text parser that is not capable of executing code. In contrast the command `eval()` is using a script parser which can execute arbitrary JavaScript code, which is a security concern.

We are not actually preventing code from entering the server, we prevent the code to be run as code. The downside of using `JSON.parse()` is that the user can submit a very large object to the database (blocking a lot of memory), with `eval()` an attacker could potentially hijack the system. There is still some concern regarding NoSQL injections, but only if password restricted user accounts are added to the portal. See Section ?? for more information.

## 4.2 Client Side - Preparation and Measurement

Our preparation & measurement relies on the profiling capabilities of Score-P. The target application can be instrumented with Score-P to produce a profile.cubex file, which contains aggregated performance data. You can read more about Score-P in Section 2.1.1, and more about how Score-P is used as part of our analysis methodology in Chapter 5.

### 4.2.1 Performance Data Parsing

The Score-P command line tool `scorep-score` is used to access the profiling data stored in the `profile.cubex` file, sample output that illustrates the data stored in the `profile.cubex` file is shown in Figure 4.2 and Figure 4.3. The times collected by Score-P in profiling mode are aggregated.

The output can be uploaded to our portal, where it will be parsed and used to further derive performance metrics and to automatically fill the fields of the corresponding application database entry. We also considered using CUBE (which is part of Score-P see Section 2.1.1) to investigate the profiles, but this approach does not give us the function group assignment of Score-P.

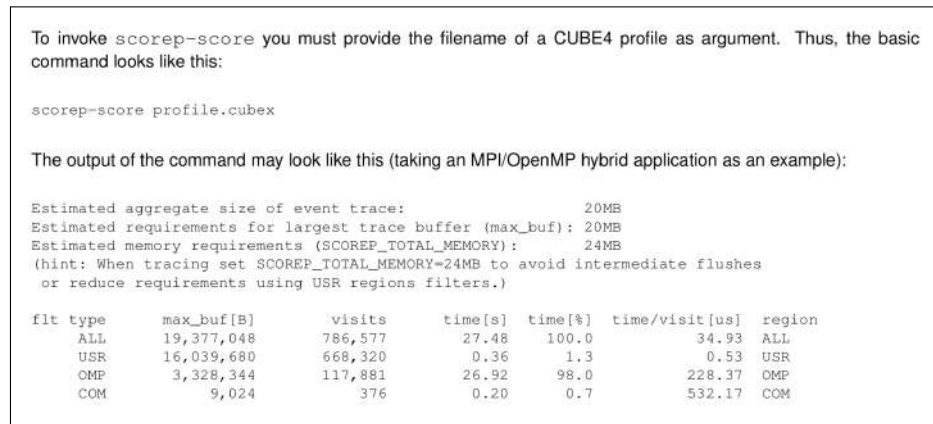


Figure 4.2: The normal output of the command line tool `scorep-score` when used on a `profile.cubex` file, reprinted from the Score-P Manual [3].

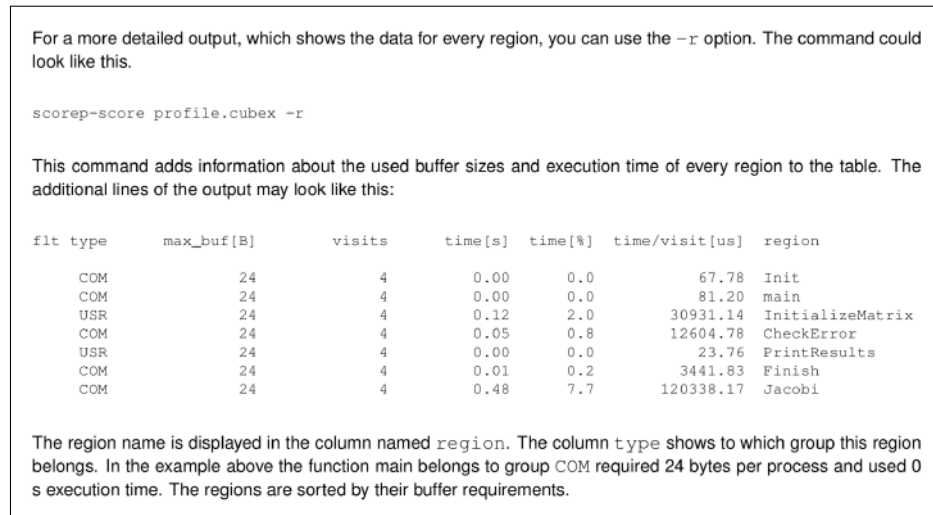


Figure 4.3: The extended output of the command line tool `scorep-score` when used on a `profile.cubex` file, reprinted from the Score-P Manual [3].

### 4.2.2 Score-P Function Groups

Score-P groups functions (regions) into function groups. The function groups that Score-P distinguishes are:

- OMP: OpenMP constructs.
- MPI: MPI functions.
- SHMEM: SHMEM functions.
- PTHREAD: Pthreads functions.
- CUDA: CUDA API functions and kernels.
- OPENCL: OpenCL API functions and kernels.
- OPENACC: OpenACC API functions and kernels.
- MEMORY: libc and C++ memory allocation and deallocations functions.
- COM: functions implemented by the user, that appear on a call-path to functions from the groups above (they are therefore considered to be part of "computation").
- USR: functions implemented by the user, except those in the COM group.
- LIB: user wrapped library functions.

As our portal is focused on the analysis of programming paradigms and parallel applications, we make some adjustments to the preset groups of Score-P. MEMORY, COM, and LIB are combined into our new USR group. On the other hand, MPI and OpenMP are extended by sub groups (e.g. OpenMP Synchronisation, OpenMP Tasking, etc.). For more information on the sub groups and their assignment please refer to Section 4.3.5.

### 4.2.3 SLURM Job Script Generator

We also developed an accompanying job script generator that completes the instrumentation, compilation, execution, and data extraction steps of the higher level Preparation & Measurement. For more information on the SLURM Job Script Generator, please refer to Chapter 5.

## 4.3 Client Side - The Application Database

The following section describes our database structure, and which performance metrics are stored. It also contains information on deriving more detailed data from a Score-P profile, and the metrics that can be stored in addition to the data provided by Score-P.

### 4.3.1 Application Database Structure

```
7 database_object = {
8   app_name: "",
9   app_version: "",
10  app_problem: "",
11  app_config: "",
12
13  total_parallel_time_s: "",
14  FLOPS: "",
15  programming_language: "",
16  IO_technique: "",
17  data_distribution: "",
18  data_replication: "",
19  workload_scheduling: "",
20
21  ALL_aggr_time_s: 0,
22  USR_aggr_time_s: 0,
23  MPI_aggr_time_s: 0,
24  SHMEM_aggr_time_s: 0,
25  OMP_aggr_time_s: 0,
26  PTHREAD_aggr_time_s: 0,
27  CUDA_aggr_time_s: 0,
28  OPENCL_aggr_time_s: 0,
29  OPENACC_aggr_time_s: 0,
30
31  MPI_P2P_aggr_time_s: 0,
32  MPI_collective_aggr_time_s: 0,
33  MPI_one_sided_aggr_time_s: 0,
34  MPI_other_aggr_time_s: 0,
35
36  OMP_work_sharing_aggr_time_s: 0,
37  OMP_synchronisation_aggr_time_s: 0,
38  OMP_tasking_aggr_time_s: 0,
39  OMP_other_aggr_time_s: 0,
40
41  MPI_communication_matrix: "",
42  OMP_schedule_clauses: "",
43
44  job_script: "",
45  comments: "",
46
47  region_text: "",
48  region_array: []
49 };
```

Figure 4.4: The database object and all fields that are stored inside the application database.

MongoDB lets you work with multiple databases, multiple collections per database, and multiple documents (application entries in our case) per collection. We follow a simple setup with one database and one collection containing all application entries.

This approach satisfies our needs for simplified database querying (we do not need to distinguish between different databases and collections). Figure 4.4 shows the structure of our database object, with all fields that get stored for each individual database entry. Name, Version, Problem and Configuration are used as database identifiers and therefore unique.

The names of the different fields are intentionally elaborate, in order to make exporting data from the database more simple and transparent without additional parsing of the content.

### 4.3.2 Metadata - Identifying Database Entries

In order to organise applications inside the database, entries are distinguished by the four identifiers Name, Version, Problem and (node) Configuration, which are unique in their combination inside the database. You can see an example of these identifiers in Section 4.4.1.

With these four identifiers, we want to give the user the possibility to keep track of different forms of the application: different application versions, problem sizes, and node configuration.

### 4.3.3 Additional Characteristics and Input Fields

In addition to the metrics collected with Score-P, the portal can also store the performance metrics listed in Figure 4.4. Some of these metrics are included in the profile generated by Score-P while fields like Data Distribution need to be entered manually. In the following list we will describe the fields that have to be manually entered by the user:

- **Total Parallel Execution Time:** The parallel execution time (wall clock time) is the time that elapses from the moment the first processor starts to the moment the last processor finishes.
- **FLOPS:** Floating-Point Operations Per Second are a measure of performance, most scientific computations require floating-point calculations.
- **Programming Language:** C, C++, Fortran, etc., or a combination of these.
- **Input / Output Technique:** There are different strategies and I/O techniques like: (a) every process writes to a local file, (b) processes write to the same file via fortran direct access, (c) the applications uses advanced MPI I/O features.
- **Data Distribution:** Data can be distributed equally or unequally among processes.
- **Data Replication:** Processes can hold a portion of the data or data can be replicated.
- **Workload Scheduling:** Processes can know their workload a priori (static), or the workload can be allocated during runtime (dynamic).
- **Communication Matrix:** The communication matrix can be submitted as a CSV to the database in order to store it together with the performance data. This matrix gives insight into communication between processes, number of messages, message sizes, etc., it can be generated by tracing the applications and using trace visualisers like Vampir.
- **OpenMP Schedule Clauses:** This field receives a CSV file with all OpenMP schedule clauses in the format: file, line, clause. It can be interesting to know which schedule clauses are used by the applications.
- **Job Script:** The job script that was used to execute the target application can give more insight into the configuration and environment variables that were used.
- **Comments:** Additional comments where the user can store information that is not part of the fields above.



### 4.3.4 Data Upload and Input Form Masking

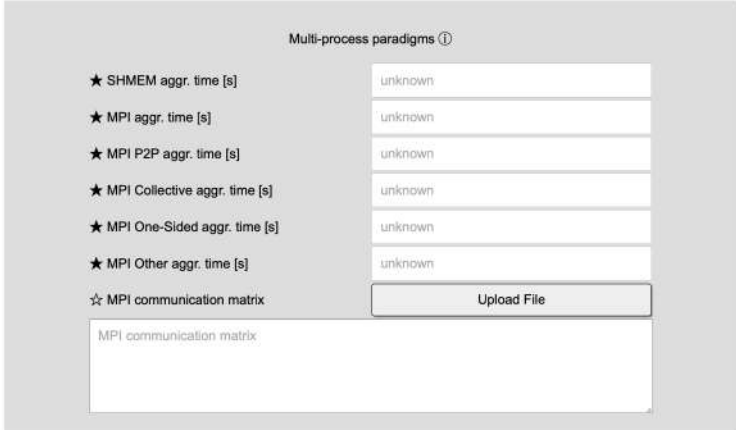
Performance data can be either be uploaded through a Score-P profile, or added manually through a graphical user interface, for more information please refer to Chapter 5. Data like the database identifiers Name, Version, Problem, and Configuration, have to be entered manually. In order to keep the uploads clean and avoid logical errors when entering data, we employed a regular expression mask for input fields, shown in Figure 4.5.

```
67
68 // a simple mask to filter user input
69 $(".input_mask").mask("XXXXXXXXXXXXXXXXXX", {
70   translation: { "X": { pattern: /[A-Za-z0-9._-]/ } }
71 });
72
```

Figure 4.5: The regular expression input mask we use for most of the input fields when manually inserting data into the application database.

Figure 4.6 shows an input form of the portal. The fields are marked with a black star if the corresponding data is included in a Score-P profile, and would be automatically filled if the profile is uploaded.

The fields for the aggregated times are limited by the input mask defined in Section 4.3.4, other fields, like the MPI communication matrix, are not limited by input masking. This does open potential problems for code injection, how our portal deals with this security issue is elaborated in Section 4.1.4.



Multi-process paradigms ⓘ

★ SHMEM aggr. time [s] unknown

★ MPI aggr. time [s] unknown

★ MPI P2P aggr. time [s] unknown

★ MPI Collective aggr. time [s] unknown

★ MPI One-Sided aggr. time [s] unknown

★ MPI Other aggr. time [s] unknown

☆ MPI communication matrix

Upload File

MPI communication matrix

Figure 4.6: The input form for the Multi-process fields of the database entry.

### 4.3.5 Programming Paradigm Assignment

We grouped MPI and OpenMP functions into sub groups by looking at their function names. By doing so, we extended the function groups provided by Score-P (introduced in Section 4.2.2) with additional sub groups for MPI and OpenMP. In the following list we describe the OpenMP sub groups that we use for our portal:

- **Work-Sharing:** These constructs divide execution of the enclosed region among threads (DO / FOR, PARALLEL, SECTION, SINGLE). They do not start new threads and have an implied barrier at the end.
- **Synchronisation:** Synchronisation constructs synchronise threads to ensure correct results, when e.g. two threads try to update the same variable at the same time (MASTER, CRITICAL, BARRIER, TASKWAIT, ATOMIC, FLUSH, ORDERED).
- **Tasking:** These constructs define an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread (TASK).
- **OpenMP Other:** This sub group contains all constructs not included in the sub groups defined above.

The sub groups of e.g. MPI are already mentioned in "A Large-Scale Study of MPI Usage" presented in Section 2.2.2 and are part of the MPI 3.1 standard. In the following list we describe the MPI sub groups that we use for our portal, the descriptions are reprinted from [25]:

- **Point-to-Point:** This feature specifies how to transmit messages between a pair of processes where both sender and receiver cooperate with each other (MPI\_Send, MPI\_Recv, MPI\_Wait, etc.).
- **Collective:** This feature describes synchronization, data movement, or collective computation that involve all processes within the scope of a communicator (MPI\_Barrier, MPI\_Bcast, MPI\_Reduce, etc.).
- **One-Sided:** This feature defines a communication where a process can write data to or read data from another process without involving that other process directly. Various synchronization models are defined, some involving all the processes in the underlying group that formed the one-sided "window" (MPI\_Accumulate, MPI\_Get, MPI\_Put, MPI\_Win, etc.).
- **MPI Other:** Contains all other MPI functions that are not included in the sub groups above (MPI\_Get\_address, MPI\_Get\_processor\_name, etc.).

### 4.3.6 Filtering and Scope Selection

Most of the visualisation provided by the portal accepts a scope (list) of database entries as input. There are two methods of filtering or selecting this scope. Either by querying the database, as shown in Figure 4.7 and Figure 4.8, or by using the Application Similarity Clustering presented in Section 4.4.5.

Given the scope selection functionality, the user can generate complex analysis by combining multiple analysis steps. For more information on how to use the scope selection, please refer to Chapter 5.

General Filters ⓘ

<= Total parallel time [s] <=   
 <= FLOPS <=   
 Programming language   
 I/O technique   
 Data distribution and replication    
 Workload scheduling

Figure 4.7: General filters for selecting a specific list of applications.

Aggregated Time % Filters ⓘ

<= User aggr. time [%] <=   
 <= SHMEM aggr. time [%] <=   
 <= MPI aggr. time [%] <=   
 <= Pthreads aggr. time [%] <=   
 <= OpenMP aggr. time [%] <=   
 <= CUDA aggr. time [%] <=   
 <= OpenCL aggr. time [%] <=   
 <= OpenACC aggr. time [%] <=

Figure 4.8: Filters for aggregated times of programming paradigms groups, for selecting a specific list of database entries.

## 4.4 Client Side - Analysis and Visualisation

The following section contains explanations on the reasoning behind comparison views provided in the analysis and visualisation component of PAP. We focus on why we chose certain visualisation, for a more detailed explanation on how to use and customize the individual charts, please refer to Chapter 5.

#### 4.4.1 Application Group Summary

Given the performance metrics that are stored in the application database, the individual summary and the application comparison focus on the aggregated time spent in high level programming paradigms, specific function groups, and in the case of MPI and OpenMP also sub groups, as shown in Figure 4.9.

We decided to use the sunburst chart of our graphing library Plotly.js, because it combines the good overview of a pie chart with multiple levels of depth for each area. The proportions of the individual areas are intentionally kept static. This was done in order to improve readability of the chart and avoid misleading representation, e.g. 0% areas would have to be omitted completely.

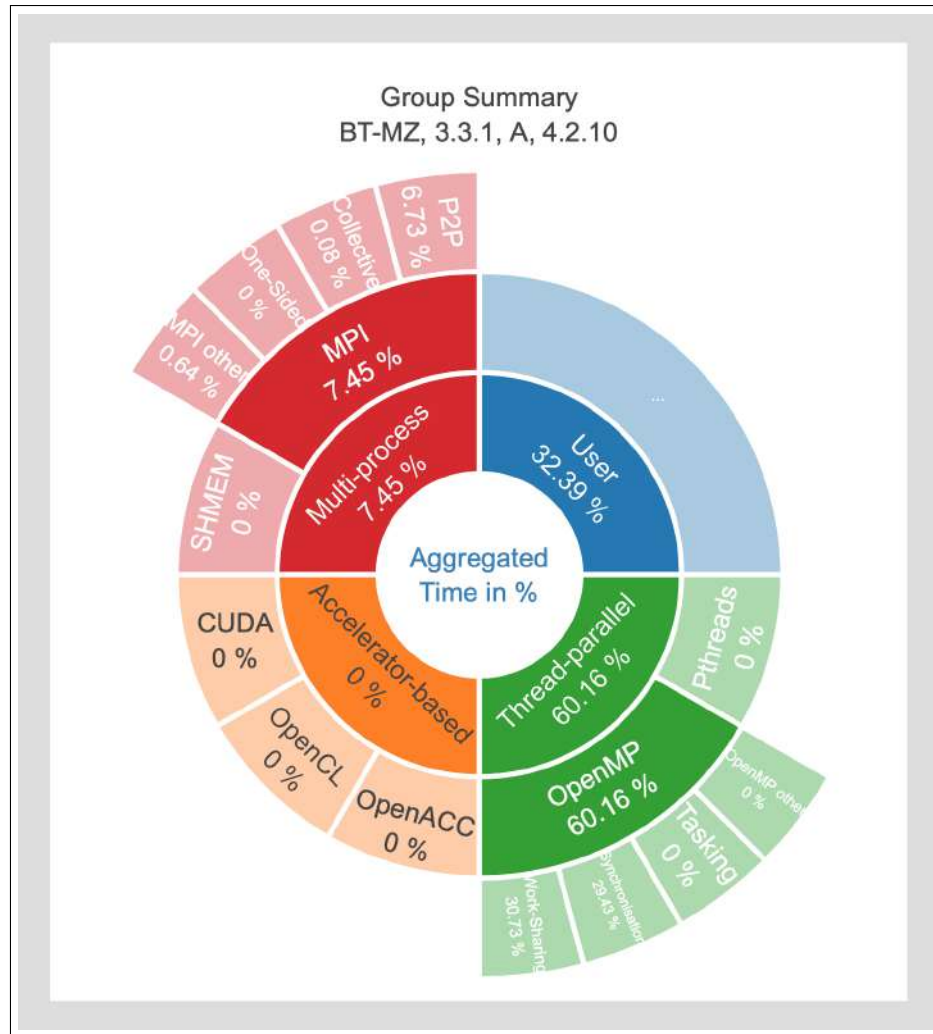


Figure 4.9: The Group Summary view for an individual applications, it provides aggregated time in % for different levels of abstraction: (a) high level groups (Multi-process), (b) individual programming paradigms (MPI), and (c) sub groups of paradigms (MPI P2P).

## 4.4.2 Application Region List

We decided to profit from the depth of a Score-P profile, by also showing the individual function (region) data of an application. Figure 4.10 shows the Application Region List, that contains group, sub group, and aggregated time about each function, construct, and region of the application.

We also implemented different forms of customization of this chart, e.g. sorting and hiding columns, for more information on the usage of this chart please refer to Chapter 5.

Aggregated time for all regions  
BT-MZ, 3.3.1, A, 4.2.10

Region name	Group	Sub Group	Time [s]	Time [%]
binvrhs	USR	-	59.51 s	12.565 %
matmul_sub	USR	-	48.39 s	10.217 %
!\$omp_implicit_barrier_@y_solve.f:406	OMP	synchronisation	46.29 s	9.774 %
!\$omp_do_@z_solve.f:52	OMP	work_sharing	46.28 s	9.772 %
!\$omp_implicit_barrier_@x_solve.f:407	OMP	synchronisation	46.01 s	9.715 %
!\$omp_do_@y_solve.f:52	OMP	work_sharing	42.95 s	9.069 %
!\$omp_do_@x_solve.f:54	OMP	work_sharing	42.35 s	8.942 %
matvec_sub	USR	-	38.32 s	8.091 %
!\$omp_implicit_barrier_@z_solve.f:428	OMP	synchronisation	36.62 s	7.732 %
MPL_Waitall	MPI	p2p	31.81 s	6.716 %
!\$omp_implicit_barrier_@rhs.f:353	OMP	synchronisation	4.64 s	0.98 %
!\$omp_do_@rhs.f:191	OMP	work_sharing	3.2 s	0.676 %
!\$omp_do_@rhs.f:80	OMP	work_sharing	3.13 s	0.661 %
MPL_Init	MPI	other	3.04 s	0.642 %
lhsinit	USR	-	2.67 s	0.564 %
!\$omp_do_@rhs.f:301	OMP	work_sharing	2.21 s	0.467 %
binvrhs	USR	-	1.91 s	0.403 %
exact_solution	USR	-	1.27 s	0.268 %
!\$omp_implicit_barrier_@rhs.f:72	OMP	synchronisation	1.25 s	0.264 %
!\$omp_do_@initialize.f:50	OMP	work_sharing	0.84 s	0.177 %
!\$omp_implicit_barrier_@rhs.f:423	OMP	synchronisation	0.83 s	0.175 %
!\$omp_do_@rhs.f:384	OMP	work_sharing	0.77 s	0.163 %
!\$omp_do_@rhs.f:37	OMP	work_sharing	0.76 s	0.16 %
!\$omp_implicit_barrier_@initialize.f:167	OMP	synchronisation	0.7 s	0.148 %
!\$omp_do_@rhs.f:62	OMP	work_sharing	0.61 s	0.129 %

Figure 4.10: The Application Region List provided by PAP, shows every function of the target application with the corresponding group, sub group, and aggregated time.

### 4.4.3 Application Group Comparison

The Application Group Comparison displays the information contained in the Application Group Summary for multiple database entries. It can be used to compare programming paradigm usage of applications.

There are multiple ways of customizing this chart, like hiding programming paradigms or summarising sub groups into higher level paradigm groups. The different entries can be sorted, and the chart can be manipulated by zooming or dragging axes. For more information on the usage of this chart, please refer to Chapter 5.

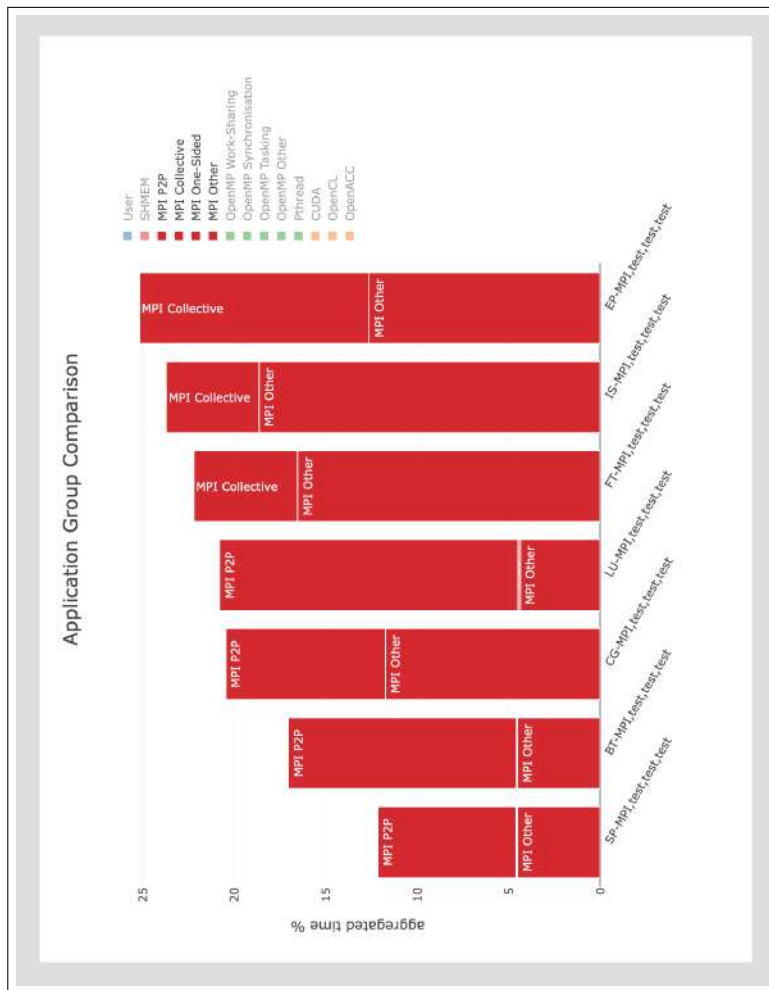


Figure 4.11: The Application Group Comparison provided by PAP, it allows the comparison of programming paradigm usage of multiple applications. In this example other programming paradigms are hidden in order to only compare the MPI sub groups.

#### 4.4.4 Programming Paradigm Statistics

Inspired by "A Large-Scale Study of MPI Usage" (see more Section 2.2.2), the portal also offers statistics about programming paradigm usage. The user can generate plots about specific function usage for all tracked programming paradigms.

This functionality is also compatible with the filtering and scope selection, enabling the user to generate usage statistics about all, or only a specific group of database entries. The actual data that is visualised are counts of e.g. MPI\_Send function usage across a scope of database entries.

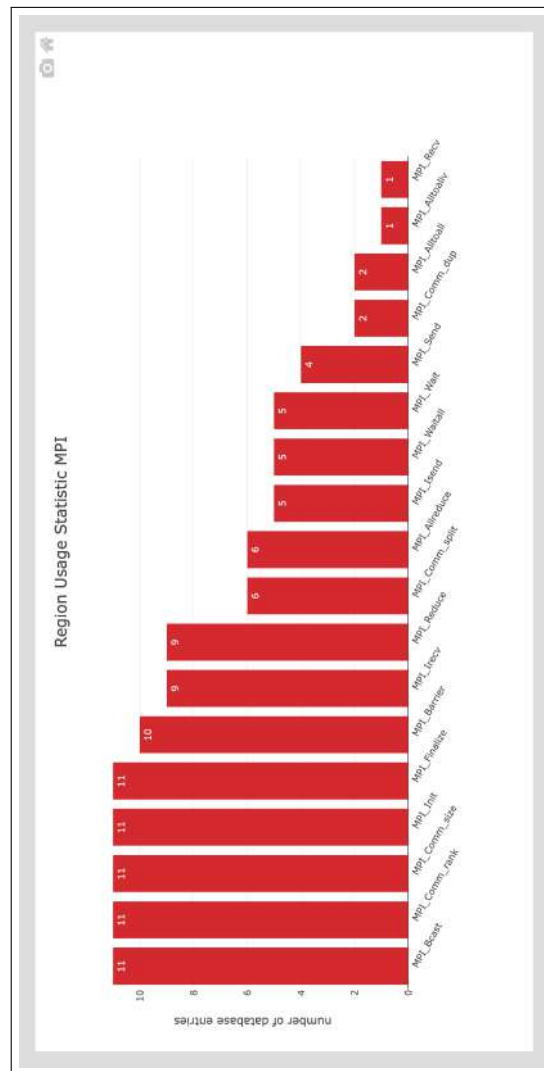


Figure 4.12: Programming Paradigm Statistics offer information about function usage across the database.

#### 4.4.5 K-Means Similarity Clustering

We rely on the one dimensional k-means clustering algorithm Ckmeans 3.4.6, by Wang et. al. [39]. The implementation that we used is provided by the JavaScript library Simple Statistics (see more Section 3.3.4).

Ckmeans was developed to solve the problem of clustering numeric data into groups with the least within group sum of squared deviations. The algorithm uses two matrices that contain values for squared deviations, which are incrementally computed, and backtracking indexes. The implementation by Simple Statistics does not automatically decide on the best number of clusters.

When the differences within groups (sum of squares) are minimized, the groups become more homogenous and the data is divided into representative groups. These representative groups emphasize differences and similarity between data.

The elements we are working with are our application database entries, and the data we are feeding to the above mentioned algorithm, is the performance data collected by Score-P. By building the groups with the least within group sum of squared deviations, we find clusters of similar applications along the chosen metric dimension.

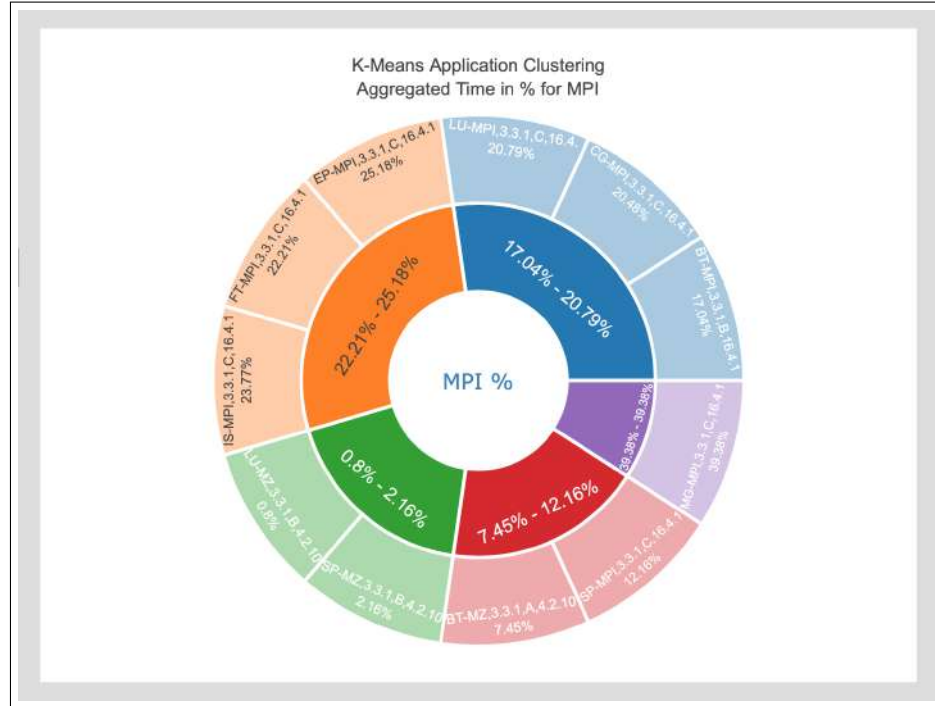


Figure 4.13: The Application Similarity Clustering provided by PAP. Database entries are grouped based on different performance metrics. The number of clusters can be chosen manually or by rule of thumb:  $\sqrt{\frac{number\_of\_entries}{2}}$ .



## Chapter 5

# Analysis Methodology and Workflow

This chapter shows the functionality of our work and can be seen as a manual for the Performance Analysis Portal for HPC Applications (PAP), it contains instructional images and example results.

### 5.1 Introducing the Analysis Methodology of PAP

Our Analysis Methodology presents a full picture of an approach to parallel performance analysis. It guides the user from preparation to analysis and incorporates the following components:

- (a) Representative performance metrics (general information like FLOPS and programming language combined with aggregated profiling data from Score-P), introduced in Section 4.3.1 and following sections.
- (b) An approach to instrumentation, compilation, execution, performance data collection, and data extraction, all within the measurement infrastructure of Score-P.
- (c) A system of organising performance data as application database entries with the identifiers Name, Version, Problem, and Configuration.
- (d) Analysis and visualisation charts that support the investigation of parallel applications, introduced in Section 4.4 and following sections.
- (e) A semi-automatic workflow with accompanying instructions that tie all of the above mentioned steps together.

We rely on the profiling capabilities of Score-P to collect most of the performance metrics for our application database. The individual summary charts, comparison charts, and clustering functionality focus on the comparison of multiple applications.

### 5.1.1 Workflow Steps - Overview

Figure 5.1 shows an overview of all pages (portal functionalities) that are part of the analysis workflow. The individual steps can be used independently. This page is also representative for all other pages, containing the top navigation bar and an area with additional information for the current page.

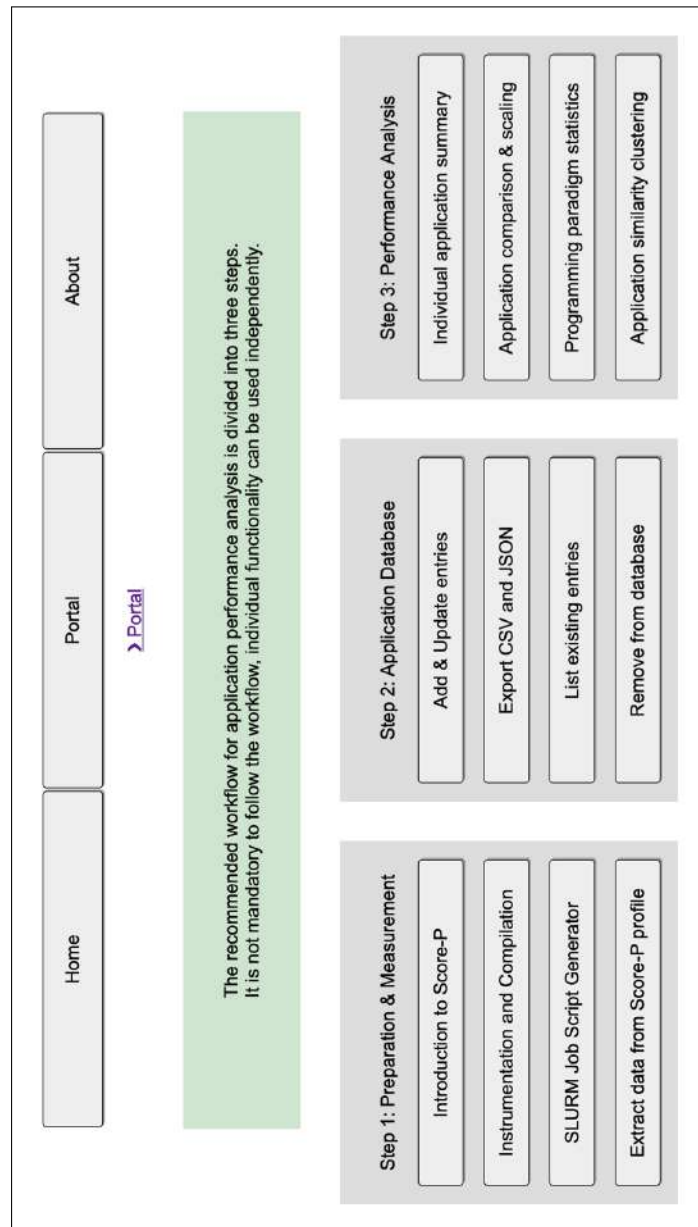


Figure 5.1: Overview page that contains links to the individual functionality provided by PAP.

## 5.2 Step 1: Preparation and Measurement

### 5.2.1 Performance Data Collection with Score-P

Figure 5.2 shows the "Introduction to Score-P" page. We rely on the measurement infrastructure of Score-P, step 1 of the analysis workflow mainly contains instructions and explanations for the instrumentation and compilation.

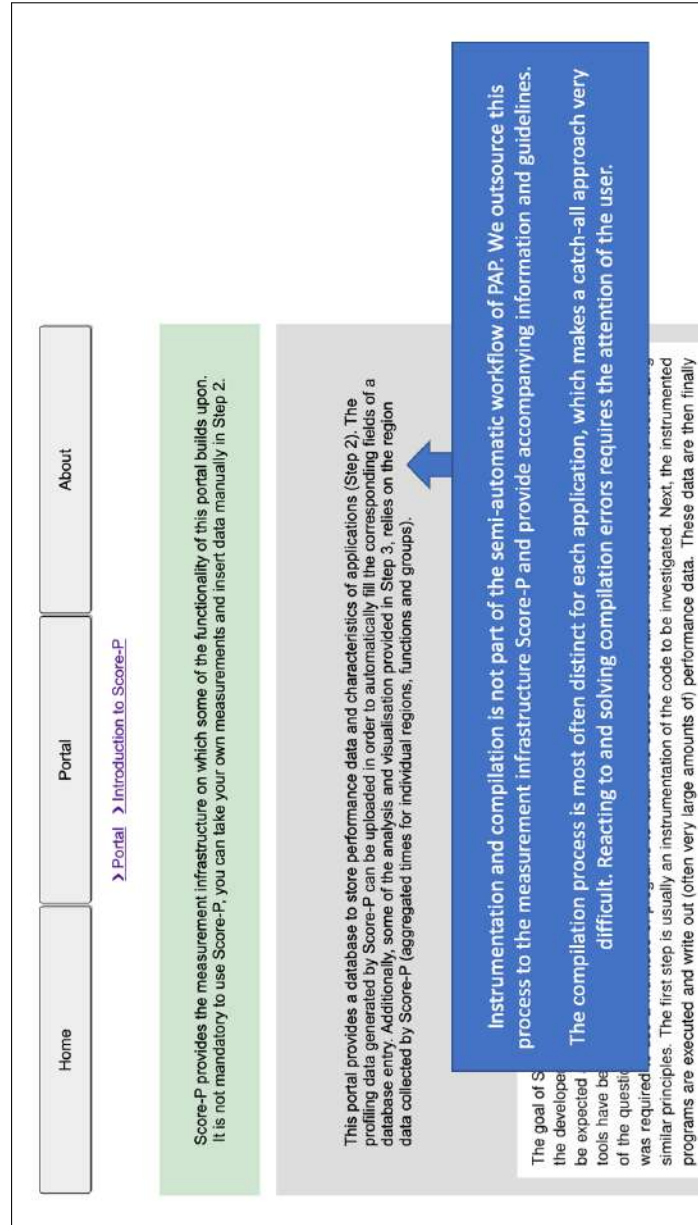


Figure 5.2: Introduction to Score-P. The Preparation & Measurement step of the portal supports instrumentation, compilation, execution, and performance data collection of the target application.

## 5.2.2 SLURM Job Script Generator

Our SLURM Job Script Generator is shown in Figure 5.3. It allows the user to generate job scripts for the his target application, in order to run them on a system working with the SLURM workload manager.

Home Portal About

Portal > [Extract data from Score-P profile](#)

The following command is used to extract region data and generate a file that the portal can read:  
`ml Score-P; scorep-score profile.cubex -r > region_data.csv;`

PAP is able to parse the output of the `scorep-score -r` command of Score-P. This command shows different performance data collected in profiling mode, with the respective group assignment.

The output of this command can then be uploaded to the portal, in order to automatically fill the corresponding fields of the application database entry.

To invoke command  
`scorep-s`

The output

Estimated memory requirements (SCOREP\_TOTAL\_MEMORY): 240MB  
 (hint: When tracing set SCOREP\_TOTAL\_MEMORY=240M to avoid intermediate flushes or reduce requirements using USR regions filters.)

Elc type	max_jbuf(mb)	visits	time(s)	time/visit(us)	region
ALL	19,377,048	786,577	27.48	100.0	34.23 ALL
USR	16,039,680	668,320	0.36	1.3	0.53 USR
OMP	3,328,344	117,881	26.92	98.0	226.37 OMP
COM	9,024	376	0.20	0.7	532.17 COM

Figure 5.3: SLURM Job Script Generator of PAP, showing the possible input fields.

### 5.2.3 Extracting Data from a Score-P Profile

Figure 5.4 shows our instruction for the extraction of data from a profile generated by Score-P. Our portal can parse the output of the **scorep-score -r** command from Score-P.

The output of the SLURM Job Script Generator, it can be directly edited in the text area.

#### Job Script Generator for SLURM ①

```
#!/bin/bash
#SBATCH -J test
#SBATCH --time=00:10:00
#SBATCH --exclusive
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --partition=xeon
#SBATCH --output=test.txt
#SBATCH --hint=nomultithread

# Please remember to load environment variables #
# e.g. export OMP_NUM_THREADS=1 #

ml Score-P
srun ./test
```

Job Name

Wall Time

Exclusive

Number of nodes

Number of processes per node

Number of threads per process

Partition

Output file

Name of executable

Figure 5.4: Extracting Data from a Score-P Profile. Generate a data format that can be uploaded to the portal, and that contains the profiling data collected by Score-P.

67

## 5.3 Step 2: Application Database Interaction

### 5.3.1 Add and Update Database Entries

Figure 5.5 shows how databases entries can be added or updated, starting with the identification process. The following subsections will contain the remaining input fields of the Add & Update step.

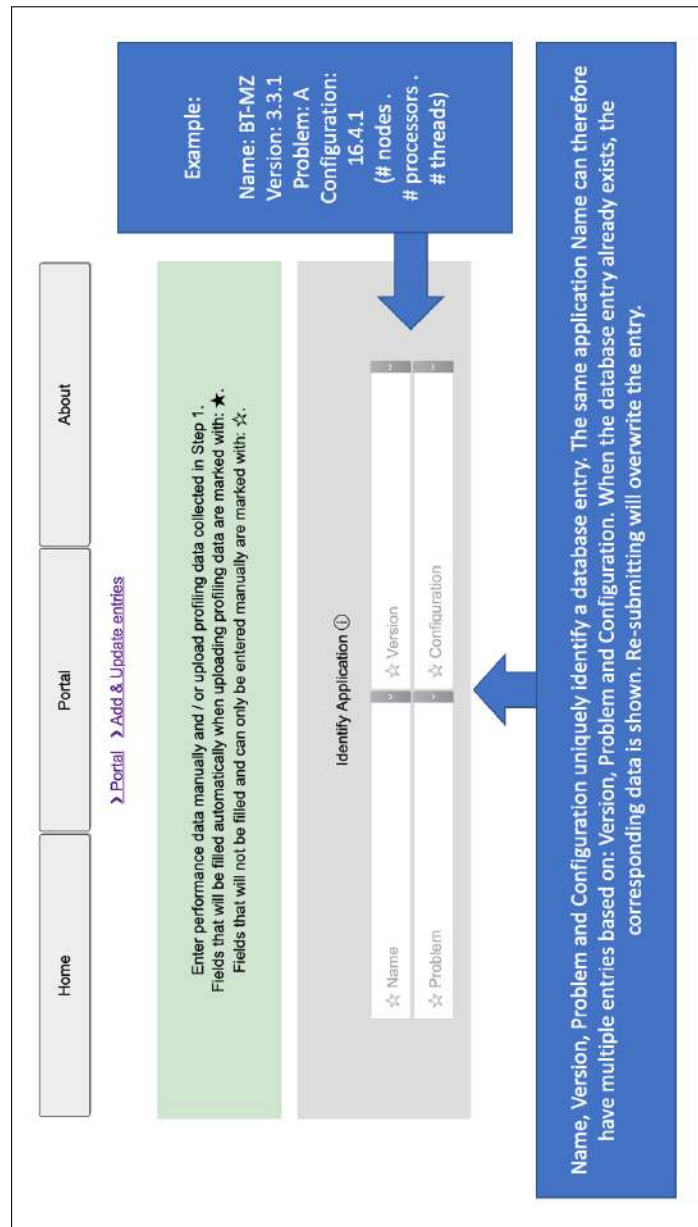


Figure 5.5: Add and Update Database Entries (1/5). The first step of adding or updating a database entry, is to identify the entry via Name, Version, Problem, and Configuration, which in their combination are unique inside the database.

The "General" Information and "Profiling Data" areas shown in Figure 5.6, are part of updating a database entry. As already mentioned in Figure 5.5, black stars mark input fields that will be automatically filled when Score-P profiling data is uploaded.

**General ①**

- ☆ Total parallel time [s]
- ☆ FLOPS (floating-point operations /s)
- ☆ Programming language
- ☆ I/O technique
- ☆ Data distribution and replication
- ☆ Workload scheduling

**Profiling Data ①**

- ☆ Upload profiling data
- ★ Total aggregated time [s]
- ★ User aggr. time [s]

**Total aggregated time:** Aggregated time is the format of our profiling data. E.g. every time a process calls a function the elapsed time of that function is aggregated.

**User region group:** Time spent in functions implemented by the user and regions that are not part of SHMEM, MPI, Pthreads, OpenMP, CUDA, OpenCL or OpenACC.

**Total parallel time:** The parallel execution time (wall clock time) is the time that elapses from the moment the first processor starts to the moment the last processor finishes.

**I/O technique:** Does every process write to a local file? Do processes write to the same file via fortran direct access? Does the application use MPI I/O features?

**Data distribution and replication:** How is data distributed among processes? Does every process hold a portion of the data or is data replicated?

**Workload scheduling:** Do the processes know their workload a priori (static), or is workload allocated during runtime (dynamic)?

Figure 5.6: Add and Update Database Entries (2/5). Different input fields for a database entry, white stars mark fields that need to be entered manually, black stars mark fields that will be filled automatically when Score-P profiling data is uploaded.

Figure 5.7 shows the Multi-process paradigm area, where data about SHMEM, MPI and MPI sub groups is collected. The MPI sub groups include: (a) Point-To-Point Communication, (b) Collective Communication, (c) One-Sided Communication, and (d) the MPI Other sub group. For more information on sub groups please refer to Section 4.3.5.

**Multi-process paradigms:**  
Regarding multi-process paradigms the portal distinguishes the groups MPI and SHMEM. MPI is further divided into the sub groups P2P, Collective, One-Sided and Other.

**Multi-process paradigms ①**

★ SHMEM aggr. time [s]	unknown
★ MPI aggr. time [s]	unknown
★ MPI P2P aggr. time [s]	unknown
★ MPI Collective aggr. time [s]	unknown
★ MPI One-Sided aggr. time [s]	unknown
★ MPI Other aggr. time [s]	unknown
☆ MPI communication matrix	Upload File

**MPI communication matrix**

**MPI communication matrix:** The communication matrix can be submitted to the database in order to store it together with the performance data.

Figure 5.7: Add and Update Database Entries (3/5). The Multi-process paradigm area for adding or updating database entries. It contains fields for SHMEM, MPI, and MPI sub groups.





Fields for the Job Script and additional comments, shown in Figure 5.9, are also part of the input form for database entries. The Region Data field contains the parsed information of profiling data collected with Score-P, in the JSON format. The Submit button, confirms all fields of the input form and sends data to the server.

**Additional Information ①**

☆ Job Script

☆ Region data

☆ Comments

**Submit to database ①**

Submit

**Job Script:**  
The Job Script field is meant for the job script used to run the application.

**Comments:**  
The Comment field can hold additional information that was not mentioned above.

**Region data:**  
Region data is content of the profiling data collected with Score-P.

Name, Version, Problem and Configuration uniquely identify a database entry. Submitting existing identifiers will overwrite the entry.

Figure 5.9: Add and Update Database Entries (5/5). The input form for adding or updating database entries also contains fields for the Job Script, Comments, and the parsed Region Data.

### 5.3.2 Export Database Entries

Figure 5.10 shows how the user can export database entries, and also a list of all different formats. Either specific fields of the database entry or the whole database object can be exported.

The following fields of the application database entry can be exported:

- Summary (CSV)
- Job Script (text)
- Comments (text)
- MPI comm. Matrix (text)
- OpenMP schedule cl. (text)
- Region Data (CSV)
- Region Data (JSON)
- Database object (JSON)

Identify Application ①

BT-MZ 3.3.1

A 4.2.10

Select export content and format

Region data (CSV)

region,group,sub\_group,aggr\_time,s\_aggr\_time\_p  
matvec,sub\_USR,-38.32,8.1  
binvcrhs,USR,-59.51,12.6  
matmul\_sub\_USR,-48.39,10.2  
binvcrhs,USR,-1.91,0.4  
lhsinit,USR,-2.67,0.6  
exact\_solution,USR,-1.27,0.3  
lsomp\_parallel @exch\_qbc.f:255,OMP,work\_sharing,0.04,0  
lsomp\_parallel @exch\_qbc.f:244,OMP,work\_sharing,0.04,0  
lsomp\_parallel @exch\_qbc.f:215,OMP,work\_sharing,0.04,0  
lsomp\_parallel @exch\_qbc.f:204,OMP,work\_sharing,0.04,0  
lsomp\_parallel @rhs.f:28,OMP,work\_sharing,0.14,0  
lsomp\_parallel @y\_solve.f:43,OMP,work\_sharing,0.03,0  
lsomp\_parallel @z\_solve.f:43,OMP,work\_sharing,0.03,0  
lsomp\_parallel @x\_solve.f:46,OMP,work\_sharing,0.03,0  
lsomp\_parallel @add.f:22,OMP,work\_sharing,0.02,0

Figure 5.10: Export Database Entries. Export different fields of database entries in different formats, or export the whole database object.

### 5.3.3 Listing and Querying Database Entries

Most of the visualisation provided by the portal, shown in Section 5.4, accepts a list of database entries as input. In order to generate such a list the user can query the database with the fields shown in Figure 5.11. Empty fields allow any value, otherwise the list will only contain database entries that match the fields.

**General Filters ①**

Total parallel time [s] <= <=

FLOPS <= <=

Programming language any

I/O technique any

Data distribution and replication any

Workload scheduling any

The following fields can be used to filter general information about database entries. You can filter unknown fields and specific values. This allows the user to generate database entry lists for the scope selection in the analysis views of step 3.

(Hint: Remember blank and zero is not the same for number range fields. Blank: the value has not been entered and is unknown. Zero: the value is known and can be zero.)

Figure 5.11: Listing and Querying Database Entries (1/2). General Filters for querying the database and generating a filtered list of entries.

Figure 5.12 shows the fields used to query based on aggregated time per programming paradigm. These range fields will only work on database entries that have been updated with profiling data from Score-P, or for entries where the respective fields have been entered manually.

The screenshot shows a web interface for filtering database entries. It features a table of filter fields and a blue information box.

Aggregated Time % Filters ⓘ		
<=	User aggr. time [%]	<=
<=	SHMEM aggr. time [%]	<=
<=	MPI aggr. time [%]	<=
<=	Pthreads aggr. time [%]	<=
<=	OpenMP aggr. time [%]	<=
<=	CUDA aggr. time [%]	<=
<=	OpenCL aggr. time [%]	<=
<=	OpenACC aggr. time [%]	<=

A large blue arrow points from the filter fields to the information box.

**The following fields can be used to filter entries with profiling data based on aggr. time [%] (precision is 000.00).**

**Only show entries with profiling data:**  
Set User aggr. time [%] to: 0 - 100

**Show entries that have both MPI and OpenMP:**  
Set MPI aggr. time [%] to: 0.001 - 100 and  
set OpenMP aggr. time [%] to: 0.001 - 100

**(Hint: Remember blank and zero is not the same for number range fields. Blank: the value has not been entered and is unknown. Zero: the value is known and can be zero.)**

Figure 5.12: Listing and Querying Database Entries (2/2). Filters for aggregated times per programming paradigm, with the purpose of querying the database and generating a filtered list of entries.

### 5.3.4 Removing Database Entries

Figure 5.13 shows the form for removing a database entry. After identifying the database entry via Name, Version, Problem, and Configuration, the user needs to confirm his intent of removing the entry, after which all corresponding data will be deleted.

The user has to identify the database entry with the Name, Version, Problem and Configuration of the target application.

Once the data is deleted, the database entry cannot be restored.

Home Portal About

➤ Portal ➤ Remove from database

Removing an entry will delete all corresponding data. This action cannot be undone.

Identify Application ①

Name	Version
Problem	Configuration

Remove entry

Figure 5.13: Removing Database Entries. How to remove an application entry from the database, and delete all corresponding data.

## 5.4 Step 3: Performance Analysis

### 5.4.1 Analysis Scope Selection

All pages that offer visualisation start with the input form shown in Figure 5.14, except for the Individual Application Summary introduced in Section 5.4.2, . A list of entries can be generated by manually adding entries to the list, or by copy pasting a list directly into the Scope area.

Home Portal About

[Portal](#) [Application comparison & scaling](#)

Compare multiple database entries with tables and plots.

Select Scope ⓘ

Name	Problem
BT-MPI, 3.3.1, B, 16.4.1	
CG-MPI, 3.3.1, C, 16.4.1	
EP-MPI, 3.3.1, C, 16.4.1	
FT-MPI, 3.3.1, C, 16.4.1	
IS-MPI, 3.3.1, C, 16.4.1	
LU-MPI, 3.3.1, C, 16.4.1	
SP-MPI, 3.3.1, C, 16.4.1	

Add Show

The Scope Selection is a functionality to quickly add a list of database entries to an analysis visualization.

The list of database entries can be generated in the step:  
> Portal > List Existing Entries.

A list of entries can also be copied from the Application Similarity Clustering, or entered manually.

Figure 5.14: Analysis Scope Selection. Scope selection for comparison views, most views accept a list of database entries as input.

## 5.4.2 Individual Application Summary

Figure 5.15 shows the Individual Application Summary, which is the only view that investigates just one application. Presented is a table containing general information about the application and a "sunburst" chart that gives an overview of the paradigm usage of the application.

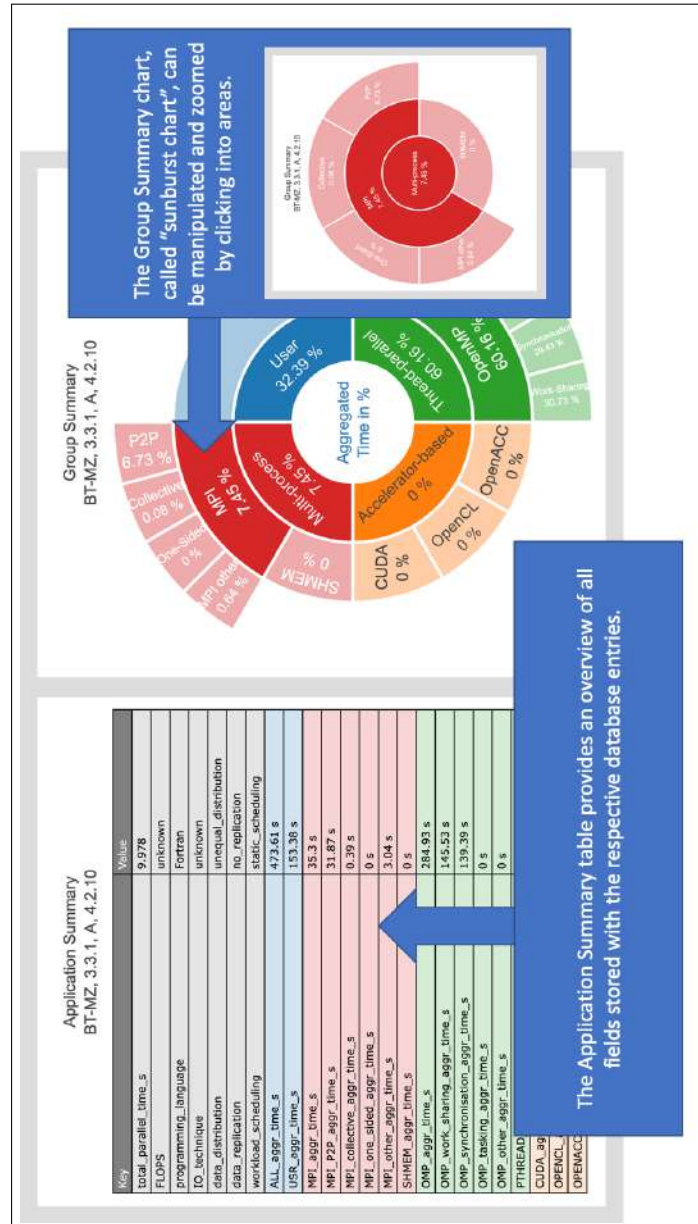


Figure 5.15: Individual Application Summary (1/2). The Application Summary and Group Summary are charts generated as part of the Individual Application Summary, and give insight into the programming paradigm usage of an application.



Figure 5.16 is also part of the Individual Application Summary and shows the Region Data for the application. Each function, construct or user implemented region is listed with the corresponding group, sub group, aggregated time and time in percent. This data is only available when the user uploaded Score-P profiling data.

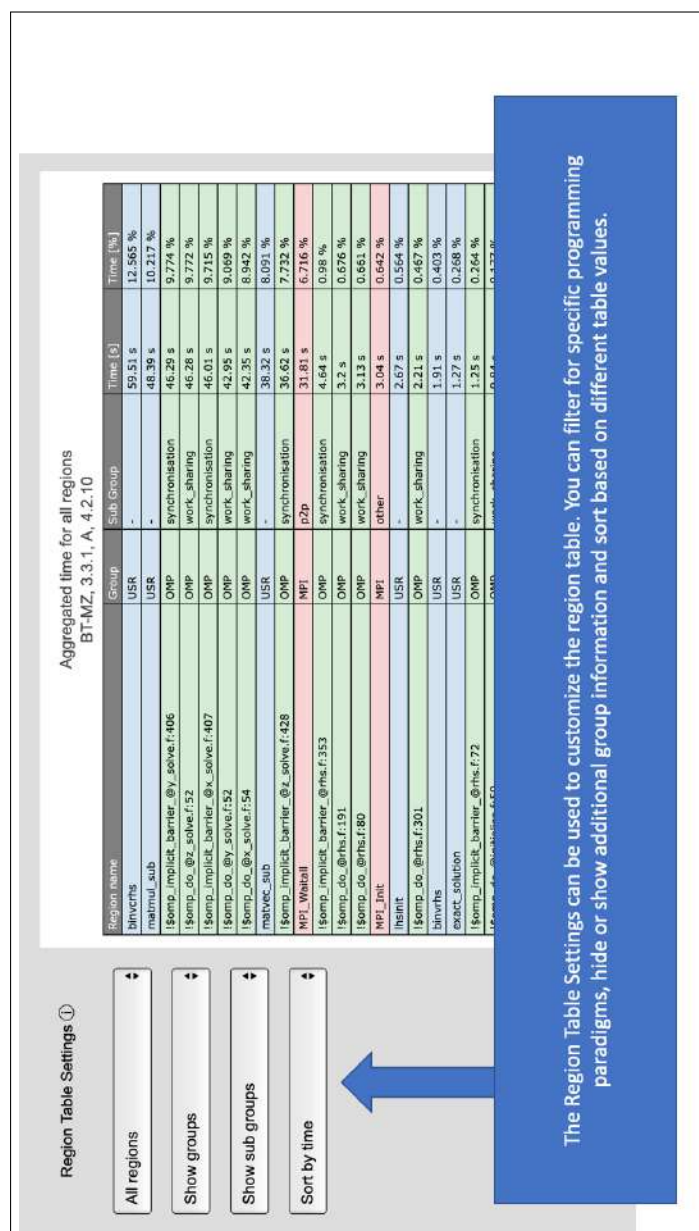


Figure 5.16: Individual Application Summary (2/2). The Region Data listing contains every function used by the application, the view can be manipulated via sorting and showing or hiding specific information.

### 5.4.3 Application Comparison and Scaling

Figure 5.17 shows the view provided by the portal for comparison of multiple applications. It focuses on time spent in programming paradigms and sub groups. The chart can be manipulated by the user, through sorting, hiding groups or sub groups, dragging axes and zooming.

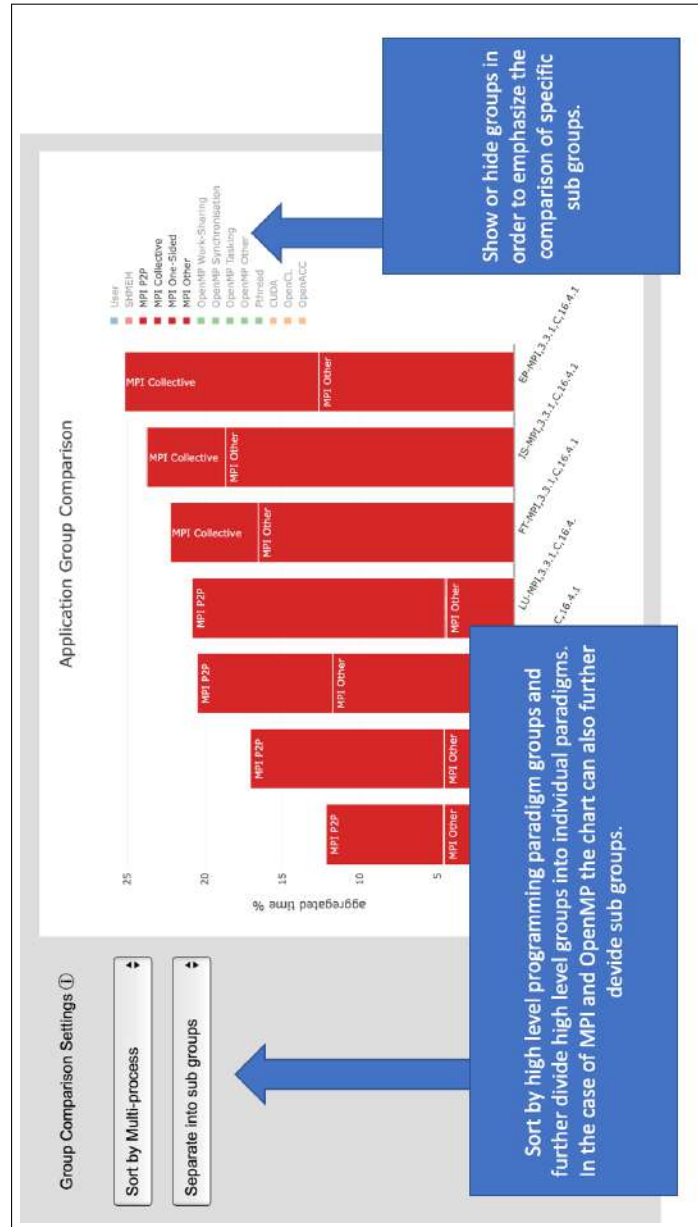


Figure 5.17: Application Comparison and Scaling. The Application Group Comparison view of PAP, provides the user with an overview of programming paradigms and sub groups for a list of database entries.

#### 5.4.4 Programming Paradigm Statistics

Statistics about programming paradigm usage are shown in Figure 5.18. The chart can be generated for each of the groups the portal can distinguish: SHMEM, MPI, Pthreads, OpenMP, CUDA, OpenCL, and OpenACC. What the chart shows is the number of applications that use a specific function or construct.

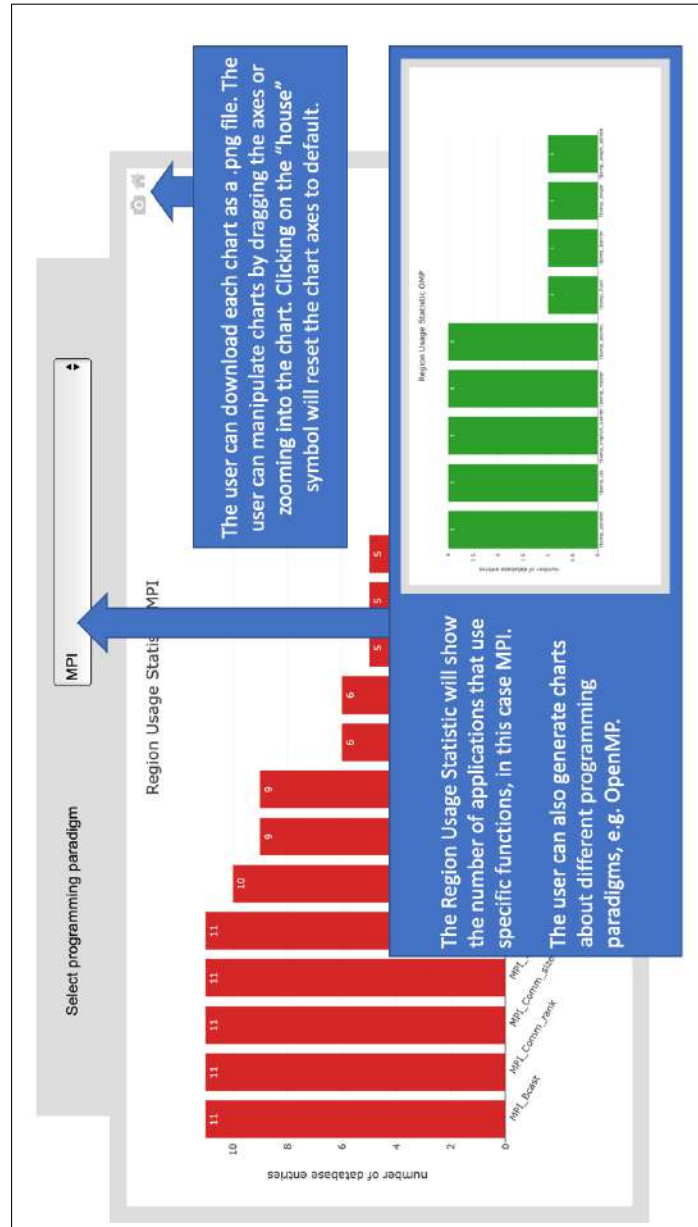


Figure 5.18: Programming Paradigm Statistics show the usage of specific functions and constructs across database entries.

### 5.4.5 Application Similarity Clustering

Figure 5.19 shows the similarity clustering based on k-means. The algorithm is explained in Section 4.4.5. The number of clusters can be adjusted, and the resulting list of clusters can be manipulated and copy pasted to other analysis views.

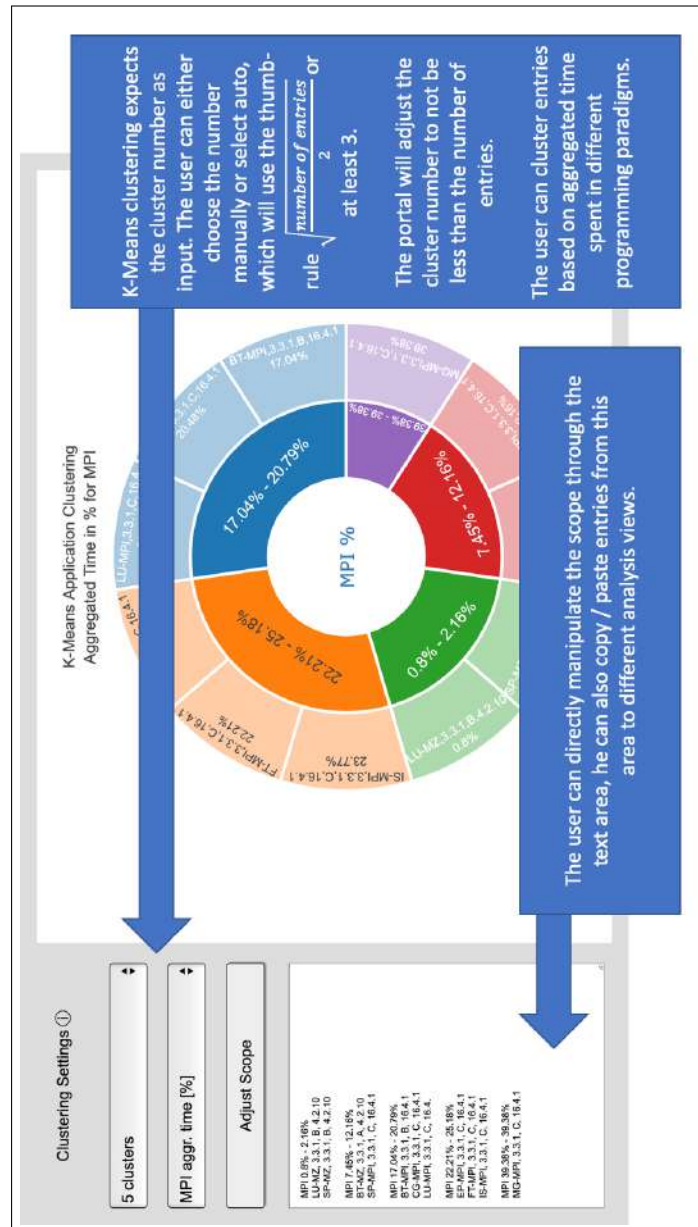


Figure 5.19: Application Similarity Clustering based on k-means, generates groups of applications based on different metrics.

## Chapter 6

# Discussion

This master thesis introduces PAP: Performance Analysis Portal for HPC Applications. PAP provides: analysis methodology, application database and performance analysis, all embedded in a web-based portal with graphical user interface.

The functionality of PAP includes: semi-automatic analysis workflow, job script generator, application database that supports querying, customizable visualisation of performance data, programming paradigm usage statistics, and application similarity clustering based on k-means. The aforementioned functionality can be used independently, but is also embedded in the analysis workflow.

Instructions and a semi-automatic workflow with a graphical user interface help the user to follow the analysis methodology. The high-level comparison of aggregated performance data is suitable to compare multiple applications. The filtering and querying capabilities of the application database support selection of candidates for performance analysis.

Additional insight beyond application performance behaviour, is gained through providing programming paradigm statistics. The statistic include region, function, or construct usage of different programming paradigms: MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, and OpenACC.

The application similarity clustering based on k-means, provides a dynamic way of grouping a high number of applications based on their aggregated time spent in the above mentioned programming paradigms. It can also help to filter groups of specific applications, e.g. with high MPI time.

The following chapter contains the Discussion section and will summarise the achieved goals, limitations, and the future work possibilities regarding PAP. We will also measure functionality provided by PAP against approaches of other studies that we presented in our related work in Section 2

## 6.1 Measuring Success

We achieved our goals formulated in Section 1.4 and successfully tackled the problems introduced in Section 1.3. PAP, the Performance Analysis Portal for HPC Applications provides:

- (a) Representative performance metrics based on data collected by Score-P and a semi-automatic analysis workflow with graphical user interface, embedded in a complete and transparent analysis methodology.
- (b) An accompanying application database that stores the collected performance data about applications, and enables filtering and querying of existing entries.
- (c) Performance analysis of individual and multiple applications, investigation of programming paradigm usage (MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC), and application similarity grouping based on k-means clustering.

### 6.1.1 Comparison of Performance Metrics

In addition to the aggregated performance data provided by Score-P, PAP also defines a list of metrics and characteristics that represent the application, as described in Section 4.3.3. The list of metrics used by PAP include: (a) FLOPS, (b) Programming Language, (c) Data Distribution, (d) Workload Scheduling, and (e) aggregated times per Programming Paradigm and sub group (e.g. MPI Point-To-Point Communication).

We compare the performance metrics of PAP with metrics defined by the related work introduced in Section 2.2.3. Joshi et. al. [22] engage in the investigation of benchmark similarity. The authors define a set of characteristics including: (a) Instruction Mix, (b) Control Flow Behaviour, (c) Instruction Level Parallelism, (d) Data Locality, and (e) Instruction Locality. These metrics are further explained in Section 2.2.3.

In contrast to PAP, the metrics defined by Joshi et. al. [22] are more sophisticated but also more complicated to collect. The authors rely on a custom-grown analyser called SCOPE, a modification of SimpleScalar [7]. An approach with metrics that only give very specific insight and that can only be collected with a custom tool, is not within the parameters of our goals. Such an approach does not satisfy our needs for a transparent methodology and an easy-to-follow analysis workflow.

### 6.1.2 Differences in Sub Groups of MPI

Our function sub group categorization explained in Section 4.3.5, was inspired by "A Large-Scale Study of MPI Usage" introduced in Section 2.2.2. Laguna et. al. [25] present an approach based on source code analysis, without actually executing the target application. PAP defines four sub groups for MPI: (a) Point-To-Point Communication, (b) Collective Communication, (c) One-Sided Communication, and (d) the MPI Other sub group that contains all remaining functions.

Compared to the four MPI sub groups of PAP, Laguna et. al. [25] use 13 MPI categories. The authors additionally distinguish: (a) Communicators, (b) Group Management, (c) MPI I/O, and (d) Error Handling to only name some of the categories. The approach of this related work is therefore more complete in contrast to PAP. On the other hand working with such a high number of sub groups makes comparison clunky, especially regarding the fact that PAP also considers many more programming paradigms than just MPI.

Differentiating between Point-To-Point, Collective, One-Sided, and the MPI Other sub group (containing the remaining functions of MPI), satisfies our goal of giving a high level overview of the application behaviour and does not distract from other programming paradigms that are also considered by PAP (SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC).

### 6.1.3 K-Means vs. Hierarchical Clustering

PAP employs an application similarity grouping based on the k-means clustering approach introduced in Section 4.4.5. The approach was inspired by the k-means clustering used by Joshi et. al. [22], see more in Section "Benchmark Similarity" 2.2.3. The authors present the results of their clustering in a table, as shown in Figure 2.21. PAP chooses a more visual approach in order to show the results of the k-means clustering, as shown in Figure 4.13. In contrast to PAP, the clustering of Joshi et. al. [22] includes one representative per group, that is closest to the center of the cluster.

A similar approach is the hierarchical clustering used by Sreepathi et. al. [34], introduced in Section "Oxbow and PADS" 2.2.1. The authors present the result of their clustering in a chart called dendrogram. In our estimation the approach we used for displaying k-means clustering results in a "sunburst" chart, Figure 4.13, is more intuitive to read compared to the dendrogram produced by hierarchical clustering, Figure 2.18. The advantage of hierarchical clustering is that every possible sub group of similar applications is included in the result.

#### 6.1.4 Comparison of Programming Paradigm Statistics

The programming paradigm usage statistics provided by PAP and introduced in Section 4.4.4, focus on the number of applications that use a specific function or construct of the target programming paradigm. Providing these statistics was inspired by "A Large-Scale Study of MPI Usage" shown in Section 2.2.2. In this work, Laguna et. al. [25] present an extensive investigation of MPI Usage. Compared to PAP the authors consider more characteristics of MPI: different MPI Standard Versions, the release date of the application source code, and they also define more MPI categories.

Laguna et. al. [25] offer more depth to their MPI usage analysis, as their work is focused on MPI. On the other hand they only offer data gather through source code analysis and omit performance metrics like aggregated execution time for MPI sub groups. The goals of PAP were defined broader with an aim on high-level characteristics about applications instead of paradigms, and not limited to only one programming paradigm.

#### 6.1.5 Other Web-Based Application Databases

PADS (Performance Analytics Data Store) [34] is a web-based infrastructure that supports collecting, storing, querying, and visualization of data. PADS is introduced in Section 2.2.1 as part of Oxbow [37] [34], which is a toolkit that offers performance metrics and an analysis workflow.

The combination of Oxbow (analysis methodology) and PADS (application database) is very similar to PAP. The performance metrics collected by Oxbow focus more on hardware counters like: scalar and vector floating-point arithmetic, scalar and vector integer arithmetic, load and store operations, etc. As already mentioned and explored in Section 6.1.3, Oxbow also employs hierarchical clustering for measuring application similarity.

Oxbow goes a step further than PAP and performs source code analysis to detect language, lines of code, number of functions, etc. Oxbow also keeps track of MPI communication, it records point-to-point and collective communication between ranks. PADS is able to visualise this communication data as part of the Oxbow and PADS workflow presented in Section 2.2.1.

The paper introducing Oxbow was published in 2013 [37] the last update was in 2014 [34], there is no public release or further mention at the time of this report in 2020, therefore Oxbow is unfortunately not available for further investigation or comparison.



## 6.2 Extensibility and Future Work

The following paragraphs contain ideas to drive forth the development of PAP, addressing some of its current limitations and opening up new approaches.

### 6.2.1 Addressing Limitations

The points mentioned in this subsection are some of the limitations of PAP.

**Automating the Compilation Process.** Compilation of the target application is currently not part of the semi-automatic workflow of PAP, and has to be done by the user. Future work could investigate approaches to automate or at least simplify the compilation process with software like EasyBuild [18].

**Collecting System Tree Information.** The current approach to parse Score-P profiles with scorep-score, loses information about the system tree (distinguishing processes and threads). Additionally parsing the profile with a tool like CUBE [24], could enrich the analysis provided by PAP.

**Storing System Statistics.** The application database can be extended to include fields for system information. The new database fields could include information about: processor type, peak performance, memory bandwidth, etc. Similar information is also collected by Oxbow, see Section 2.2.1.

**Database Backup System.** Beyond the backup provided by the production server, the application database is missing a sophisticated backup system that keeps track of entry versions and prevents unintentional deletion or manipulation.

### 6.2.2 Further Development

There is room for further development of PAP, in order to enrich the analysis and user experience.

**Extending the Subgroup Assignment.** Sub groups can be defined for the other programming paradigms (SHMEM, Pthreads, CUDA, OpenCL, and OpenACC), and the existing sub groups of MPI and OpenMP can be extended.

**User and Group Accounts.** User accounts can help organize data, e.g. by only showing the entries of specific users. Accounts can also help to restrict certain functionality (like removing database entries) and eventually open the portal to a broader audience.

**Communication Pattern Recognition.** Automated pattern recognition in MPI communication matrices would help to better understand parallel communication behaviour.

**Automated Source Code Analysis.** Our approach can be extended to include automated source code analysis to detect language, lines of code, number of functions, OpenMP schedule clauses, etc.

# References

- [1] BSC Homepage, Paraver Overview. <https://tools.bsc.es/paraver>.
- [2] HPCToolkit Homepage. <http://hpctoolkit.org>.
- [3] Scalable Performance Measurement Infrastructure for Parallel Codes - Manual v6.0. <http://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/>.
- [4] TU Dresden Compendium for Vampir. <https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/Vampir>.
- [5] Vampir 9.6 Tutorial. <https://vampir.eu/tutorial/manual>.
- [6] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.
- [7] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, (2):59–67, 2002.
- [8] David H Bailey. NAS parallel benchmarks. *Encyclopedia of Parallel Computing*, pages 1254–1259, 2011.
- [9] Robert Bell, Allen D Malony, and Sameer Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *European Conference on Parallel Processing*, pages 17–26. Springer, 2003.
- [10] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer, 2010.
- [11] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering*, (1):46–55, 1998.
- [12] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In *PARCO*, volume 22, pages 481–490, 2011.

- [13] Markus Geimer, Pavel Saviankou, Alexandre Strube, Zoltán Szebenyi, Felix Wolf, and Brian JN Wylie. Further improving the scalability of the Scalasca toolset. In *International Workshop on Applied Parallel Computing*, pages 463–473. Springer, 2010.
- [14] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.
- [15] William Gropp, William D Gropp, Argonne Distinguished Fellow Emeritus Ewing Lusk, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.
- [16] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.
- [17] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.
- [18] Kenneth Hoste, Jens Timmerman, Andy Georges, and Stijn De Weirtdt. Easybuild: Building software with ease. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 572–582. IEEE, 2012.
- [19] Kevin A Huck and Allen D Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 41. IEEE Computer Society, 2005.
- [20] Anil K Jain, Richard C Dubes, et al. *Algorithms for clustering data*, volume 6. Prentice hall Englewood Cliffs, 1988.
- [21] Ian Jolliffe. *Principal Component Analysis*. Springer, 2011.
- [22] Ajay Joshi, Aashish Phansalkar, Lieven Eeckhout, and Lizy Kurian John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.
- [23] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The Vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.
- [24] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-p: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

- [25] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. A large-scale study of MPI usage in open-source HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.
- [26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.
- [27] Gabriel Marin, Jack Dongarra, and Dan Terpstra. MIAMI: A framework for application performance diagnosis. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 158–168. IEEE, 2014.
- [28] Bernd Mohr, Allen D Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, 2002.
- [29] Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and analysis of MPI resources. 1996.
- [30] Leonid Oliker, Andrew Canning, Jonathan Carter, Costin Iancu, Michael Lijewski, Shoaib Kamil, John Shalf, Hongzhang Shan, Erich Strohmaier, Stephane Ethier, et al. Scientific application performance on candidate petascale platforms. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–12. IEEE, 2007.
- [31] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, volume 44, pages 17–31. IOS Press, 1995.
- [32] Rolf Riesen. Communication Patterns [message-passing patterns]. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.
- [33] Sameer S Shende and Allen D Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.
- [34] Sarat Sreepathi, Megan L Grodowitz, Robert Lim, Philip Taffet, Philip C Roth, Jeremy Meredith, Seyong Lee, Dong Li, and Jeffrey Vetter. Application characterization using Oxbow toolkit and PADS infrastructure. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, pages 55–63. IEEE Press, 2014.
- [35] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

- [36] Jeffrey Vetter and Chris Chembreau. mpiP: Lightweight, scalable MPI profiling. 2005. <http://mpip.sourceforge.net>.
- [37] Jeffrey S Vetter, Seyong Lee, Dong Li, Gabriel Marin, Collin McCurdy, Jeremy Meredith, Philip C Roth, and Kyle Spafford. Quantifying architectural requirements of contemporary extreme-scale scientific applications. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 3–24. Springer, 2013.
- [38] Jeffrey S Vetter and Andy Yoo. An empirical performance evaluation of scalable scientific applications. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 16–16. IEEE, 2002.
- [39] Haizhou Wang and Mingzhou Song. Ckmeans.1d.dp: Optimal  $k$ -means clustering in one dimension by dynamic programming. *The R Journal*, 3(2):29–33, 2011.



Universität  
Basel

Philosophisch-Naturwissenschaftliche  
Fakultät



## Erklärung zur wissenschaftlichen Redlichkeit (beinhaltet Erklärung zu Plagiat und Betrug)

Masterarbeit

Titel der Arbeit (*Druckschrift*):

PAP: Performance Analysis Portal for HPC Applications

Name, Vorname (*Druckschrift*):

Jakobsche, Thomas

Matrikelnummer:

2011 059 110

Mit meiner Unterschrift erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe.

Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Diese Erklärung wird ergänzt durch eine separat abgeschlossene Vereinbarung bezüglich der Veröffentlichung oder öffentlichen Zugänglichkeit dieser Arbeit.

☐ ja ☒ nein

Ort, Datum:

Lörrach, 14.02.2020

Unterschrift:

*Dieses Blatt ist in die Bachelor-, resp. Masterarbeit einzufügen.*