Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 207 (2022) 1067–1076

Procedia Computer Science

www.elsevier.com/locate/procedia

26th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2022)

Automated performance analysis tools framework for HPC programs

Maximilian Keiff[a,*], Frederic Voigt[a,*], Anna Fuchs[a], Michael Kuhn[b], Jannek Squar[a], Thomas Ludwig[c]

[a] *Universität Hamburg, Bundesstraße 45a, Hamburg 20146, Germany*
[b] *Otto von Guericke University Magdeburg, Universitätsplatz 2, Magdeburg 39106, Germany*
[c] *DKRZ GmbH, Bundesstraße 45a, Hamburg 20146, Germany*

Abstract

In the high performance computing (HPC) field, computer and natural scientists work together to solve complex problems. While computer scientists try their best to meet the needs of domain experts, the latter ones still require deeper know-how in informatics to bring their science onto machines in an optimal way. Applications typically require significant optimisation and tuning efforts to harness the performance capabilities of HPC systems. A wide range of very different analyse tools are available for this purpose, but their use is time-consuming and thus also a financial burden. This work is the first to introduce an extensible framework which does not only provide a convenient graphical interface for HPC performance analysis tools, but also simplifies their usage extremely. On this path the foundation for automated scientific software analysis and optimisation workflows is layed, which is believed to become increasingly necessary.

Keywords: optimisation tools; performance analysis; user-friendly workflow; HPC

1. Introduction

As technology advances, the possibilities increase, but so do the challenges and problems. Almost every natural science benefits from informatics technologies – especially calculation-intensive experiments and real-world simulations leverage powerful supercomputers that push the power and complexity of resource management to the extreme.

In the high performance computing field, computer and natural scientists come together. The former deploy and maintain the systems, develop system and middleware programs and support all workflows. The latter write code for science experiments (e.g. computation-heavy simulations) and run their applications on parallel systems. For an optimal porting of scientific calculations by domain experts, they still require deeper computer science knowledge.

* Corresponding author
E-mail addresses: maximilian.keiff@studium.uni-hamburg.de (Maximilian Keiff)., frederic.voigt@studium.uni-hamburg.de (Frederic Voigt).

1877-0509
10.1016/j.procs.2022.09.162

Writing correct sequential code is already challenging, while its parallelisation, debugging and optimisation demand the utmost of natural scientists, who usually lack appropriate training [6].

A study [16] showed that less than 20% of surveyed researchers use profiling tools when planning optimisations; a third of respondents are unaware of profiling tools, and about a quarter have heard of them but never used them. The domain experts were very skeptical and thought they knew better how the application behaved and where the time was spent, or they did not think a tool would help them. Another study [14] describes a very important criterion for tools to be accepted by the scientific community: having a minimum of technical details, since scientists are busy enough with the technicalities of their own science.

When looking at the system costs, it can be seen how large the waste can be when running poorly optimised programs. For example, a computing centre located at rank 33 of the Top500 list[1] in 2016 costs about € 41,000,000 with an additional € 2,000,000 per year for electricity[2]. Even small optimisations of a few percent of runtime or resource usage can have a significant impact on the business. Large computing centres have dedicated user support departments that spend up to half a year on intensive code analysis and optimisation of an application. Such staff costs around € 80,000 per person per year[2]. Any automation of the processes can significantly improve the progress.

There are already a variety of analysis tools for optimising programs on HPC systems, but these also come with a number of challenges, which are discussed in more detail in Section 2.2. These challenges with using the tools lead to a lack of optimisation, resulting in suboptimal codes that cause financial losses and do not fully exploit the research potential. In this work, the research goal how to simplify and standardise the use of the analysis tools was explored in order to overcome these problems. Thereby, the focus was not only on designing a graphical user interface, but on developing an extensible framework that lays the foundation for a fully assisted optimisation workflow.

Compared to similar approaches, which are explained in more detail in Section 3, the framework does not replace individual tools, but offers a novel, uniform interface for the use of all existing tools. The installation and operation of these tools is completely automated and the users are supported in choosing the appropriate tool for their needs.

In the following the background of HPC systems and the application of optimisation tools will be introduced. After this, similar and related approaches to tool collections as well as automatic optimisation approaches will be discussed. This is followed by the structure of the framework and the data models that enable an abstract integration of the tools. Finally, an overview of further improvements and developments towards automated scientific software analysis and optimisation workflows is given.

2. Background

HPC system architectures grow and become more complex and heterogeneous to satisfy the demands of more complicated use cases. The basic hardware and software components that are most relevant to users are described in the following.

2.1. High Performance Computing

An HPC system consists of (up to) thousands of servers – called nodes – equipped with (up to) millions of processor cores, and a high-speed network interconnecting them. In order to store the calculation results, they are written to large storage arrays consisting of hard disk drives and solid-state drives. Data is accessed in a parallel file system and has to be transferred between storage and client nodes.

These systems commonly provide a couple of head or login nodes, which are the only gateway for users to log into the system. Users require a console terminal for access via SSH. The general HPC architecture is shown in Figure 1. Many of the HPC systems operate on Linux or derivates so that the users usually face some kind of command-line interface. HPC systems hardly ever have graphical interfaces, which are only accessible with additional settings anyway.

To run the application on remote computing nodes, the users have to write additional jobscripts for the specific job scheduler. The scripting language is mostly shell or variations with additional scheduler-specific prefixed comments

[1] List of 500 fastest supercomputers worldwide: https://top500.org/

for resource control. One of the most popular schedulers is SLURM, which offers not only 27 parameters merely for submitting jobs, but also a dozen other functions [13].

The software used in an HPC system is usually organised in a package manager like Spack or EasyBuild. The user has to know which libraries and versions are required and load them with specific commands.

Since most massively parallel applications are written in C/C++ or Fortran, they often cannot benefit from modern and easy-to-use tools for object-oriented or scripting languages. Tools for parallel applications are even more limited in terms of user-friendly interfaces.

A life-cycle of large supercomputers typically lasts 4–8 years[2], which means that all systems and software components may be completely different on the next system and users are forced to dive into the new software again. Moreover, coded optimisations may no longer be suitable for the new system and require new investigations into new architecture and new features.

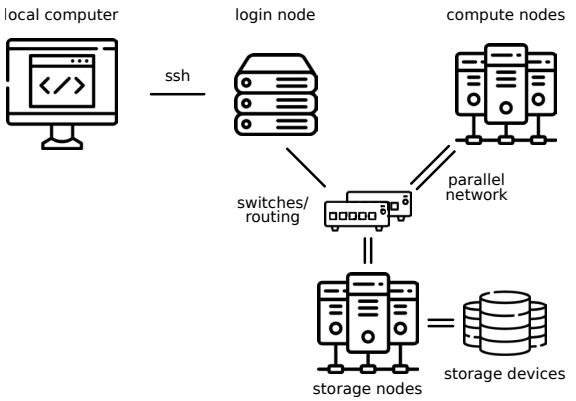

Fig. 1: A general architecture of an HPC supercomputer.

Scientific software running on HPC systems is often based on decades-old codes and is very poorly optimised [12]. Many jobs still run on single nodes or even the improper shared login node. They have large and performance-critical sequential sections (like input/output), lack modern features (vectorisation or compression [9, 4]) or missing support for specific hardware like GPUs and can therefore waste resources. Due to the observed decline of acceleration of the computational power [15], efficiency of the software will become even more critical. The successor of the previously 33rd-ranked supercomputer has doubled the power consumption, but only quadrupled the performance, which cannot keep up with the rising resource requirements of the scientific software[2].

2.2. *Optimisation Tools in HPC*

A tool-based approach can help to unravel the complexities of scientific software. There are various analysis tools that can provide the user with information about the performance of their program and thus help to uncover bottlenecks. With information about problem areas, the user is given an indication of where the manual optimisation process can begin.

Most tools fall into the category of profilers and tracers. Profilers show the course of a metric (for example, CPU consumption) over the runtime, broken down by program section. Tracers on the other hand store certain events of a metric with a timestamp and are particularly suitable for displaying information with a higher degree of abstraction. This includes, for example, information about the parallelisation of the program. Most of the tools get their information about the work of the processors by using the “hardware performance counters” on which performance relevant events are put out.

Other tool categories provide information about the present hardware. This ranges from simple information about available processors to abstract representations such as integrated performance benchmarking with respect to the provided hardware.

The control is mostly command line based, some of the tools provide GUIs and almost all of them use their own visualisation tools which are partly interoperable. In more complex use cases, it may be necessary to manually extend the own code with a tool API. The output formats are mostly different and can also be partially converted into each other.

The use of each tool is complex and usually requires familiarisation with their extensive documentation. The installation processes can be time-consuming, differs from environment to environment and is error-prone due to specific dependencies. While the installation processes have a similar scheme, the handling of the respective tools

[2] Thanks to the administration and application support departments of *Deutsches Klimarechenzentrum GmbH* for providing this data.

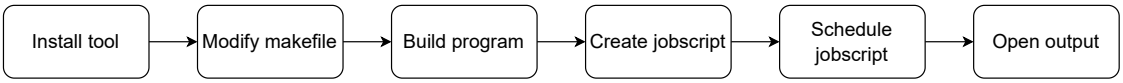

Fig. 2: Application process of an analysis tool for a program in the HPC context.

Feature \ Tool	TAU	HPC-Toolkit	Score-P	Extrae	Darshan	VTune	ITAC	Intel Advisor
Profiling	X	X	X	indirect	X	X	X	X
Tracing	X	X	X	X	X	X	X	X
Sampling	X	X	X	X		X	X	X
Instrumentation	X		X	X	X	X	X	X
PAPI events	X	X	X	X			X	
API	X	X	X	X		X	X	X
MPI	X	X	X	X	MPI-IO	X	X	X
OpenMP	X	X	X	X		X	X	X
Browser/GUI	X	X	X			X	X	X
Command line	X	X	X	X	X	X	X	X
x86, Power, ARM and GPU	X	X	X	X				
Input/Output View	X	X	X	experimental	X	X	restricted	

Table 1: Comparison of the features of some selected analysis tools.

varies significantly. Almost all tools are operated differently and require either compiling the program in a special way, using environment variables, specific commands or further post-processing tools.

Since almost every tool offers unique features, it is desirable to carry out jobs with several different tools. Once a domain expert has become familiar with one tool, the required time and the potential for errors are almost as high for the next. Running a scripted set of jobs is also non-trivial, as it is tedious to change the tools, recompile the program, or even respond to changes in computing resources. When a program run has been successfully analysed with one tool, it is not trivial to subsequently examine the output with the appropriate visualisation tool.

An abstract sequence of the steps that have to be carried out for the use of a tool is shown in Figure 2. The tools abstract the lower level functions of the hardware components in addition to the specific operating system libraries. As described in [8], some of the tools have additional support for GPU computing. An overview of the mapping of highly abstract functions to some tools is given in Table 1, which exemplifies their variety.

3. Related Work

The approach in [3] has a very similar motivation to this work and aims to provide the framework *Timemory* for performance measurement and analysis. As a result, there is another code base written in C++ that claims to be more generic, efficient, and easier to use, replacing existing tools. This work however introduces a wrapper for them that allows to be much more flexible in maintenance and the assumption is made that reusing the well-known tools with a new shape is a way that can be rather accepted by both, middleware developers and users.

VI-HPS [17] is a virtual project for high-productivity supercomputing, which aims for improving the quality and acceleration of the development process of scientific codes on highly parallel computer systems. They offer trainings, organise symposia and have created a tools overview with appropriate guides on their website. It already helps scientists a lot in finding the way to the tools they need, but does not save the effort in using them.

In [11], the authors describe the need for a holistic software stack for power and energy management interfaces and their end-to-end autotuning. This work focuses on a unique solutions for different heterogeneous system layers in PowerStack, a stack of hierarchical software and firmware components in HPC fields. Although it does not address the application side, the work still shows the need for autotuning and homogenisation of different layers in order to improve the utilisation efficiency.

In [18] a tool is presented for the examination of programs for a certain environment on the basis of a compiler and profiler analysis. Interesting for this work is the demonstrated possibility to integrate different benchmarking programs, which can then be controlled via a GUI.

In Section 5.1, the possibilities for automatic improvement of the programs by the framework are described. This so-called auto tuning comes in many forms and with many approaches. Some of them are collected in [5].

The GPA (The GPU Performance API) tool [7] addresses the future need for automated optimisation support for graphic card-based codes. First, inefficiencies are identified through low-level abstract analysis and traced back to their causes. This root cause analysis is abstracted by further post-processing and finally checked against predefined rules. With the help of these rules, optimisation proposals can be made to the user. In addition to code optimisations, there are parallelisation optimisations where the hints also refer to less abstract aspects such as the rearrangement of blocks and threads.

Since it is also planned that machine learning (ML) aspects will be integrated in this presented framework, it should be mentioned that there is already application of machine learning in the HPC domain for various problems. For example job auto-scheduling with a focus on energy efficiency [2] or prediction and optimisation of applications based on data collected by performance counters [10].

4. Design and Implementation

The main goal in creating a framework that combines as many HPC tools as possible is usability and extensibility. The aim is to hide the technical hurdles and provide intuitive graphical interfaces with additional hints wherever needed. Important parameters are brought to the surface which often remain unknown and unused otherwise. The framework provides additional adjustable interfaces for advanced users.

Frontend. The user interacts with the framework through a GUI that is accessible via a webbrowser. Two different views of the usage are possible – either the user chooses a set of tools to be applied to the program or a set of more abstract aspects to analyse. They can be of a general nature like runtime and memory usage or more specific like e.g. vectorisation, shared memory communication or I/O pattern. The framework would then map these to specific tools and proceed with the specific tool configuration. Currently, the former approach is implemented.

The tool installation on the cluster which has the potential for errors is bypassed. Furthermore, an intuitive browser-based GUI on the user's computer can be used, instead of a limited remote GUI to the cluster. This is where commands are sent to the remote cluster, files are created, and output data is managed. For tool-based program examination, it is no longer necessary for the user to connect to the cluster separately, as it is also possible to load programs and associated data directly onto the cluster via the framework. All relevant specifications of the respective cluster are determined and displayed in the GUI, which includes, for example, possible partitions of the cluster for computing, programs suitable for execution and a list of all provided tools.

Tool Configuration. To achieve the goal of easy extensibility and integration of new tools, an abstract representation of the tools on the software side was devised. The use of each tool was divided into four aspects: The installation of the tool itself, the compilation of the program to be examined, the creation of a jobscript and the processing of the output. The configuration model shown in Figure 3 allows these aspects of a tool to be abstracted for use within the framework in a consistent format.

The framework can read from these configuration files and implement the necessary requirements. Through the configuration model, the extension of the framework with new measurement options of the tools, correction/modification of existing tools or even new tools is possible without changing the source code. In addition, installation aspects can be adapted for the respective hardware.

Tool Settings. A tool is displayed to the user in terms of the measurement options it offers, which are referred to as different "settings". For example, the ability to read cache misses can be a setting. Within a setting, all aspects required to implement the setting and perform the measurement are defined.

For some tools it may be necessary to configure/install them specifically for certain measurements. For this purpose, the installation specifications such as dependencies or versions are defined.

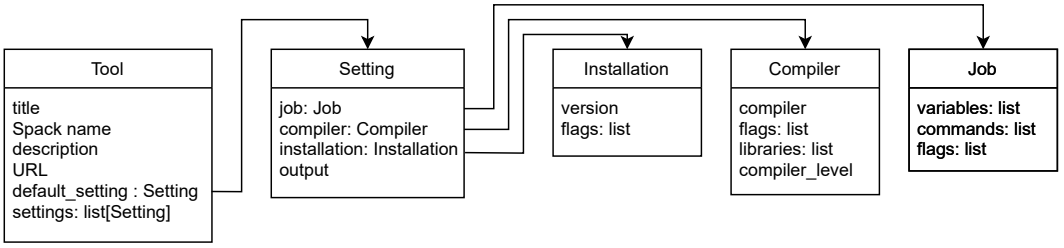

Fig. 3: Abstract model of the configuration format for the tools.

For the compilation of the target program, a Makefile is expected, which is defined in a uniform format as shown in Listing 1. Future extensions could also support other Makefile formats or build concepts as well by adding alternative pipeline branches.

```
CC = <Compiler>
# Compiler flags, paths and libraries
CFLAGS = <Flags...> -I<Include_path> -L<Library_path> -O<Optimisation_level>
LFLAGS = $(CFLAGS)
LIBS   = <Libraries...>
```

Listing 1: Structure of a Makefile with parameters denoted by the chevrons.

According to the requirements of the tool, the compiler, the compiler flags, the libraries, the optimisation level of the compiler as well as the include and linking paths can be adjusted.

Job Submission. The framework controls the execution of the program run and thus the measurements through a script that is passed to a workload manager. Currently, the SLURM scheduler is supported, but the interface can be extended to include any other scheduler. Jobscripts are created for each measurement in a structured way according to the specifications of the workload manager. An example of a jobscript that the tool could generate instead of the user doing it manually can be seen in Listing 2.

```
#!/bin/bash
#SBATCH -J jobscript
#SBATCH -o ./%x.%jobscript.out -e ./%x.%jobscript.err
#SBATCH -D ./execution
#SBATCH --cpu-bind=mask_cpu:0x1,0x4, 0x8
#SBATCH --hint=memory_bound
#SBATCH --mail-type=ALL --mail-user=scientist@university.org
#SBATCH --time=24:00:00 --no-requeue
#SBATCH -N 4 -n 4
#SBATCH --export=NONE
#SBATCH --get-user-env
#SBATCH --account=project_id --partition=cluster_partition --network=system
#SBATCH --threads-per-core=4
#--constraint="[(knl&snc4&flat)*4&haswell*1]"

mpirun -np 16 ./sample-program 7 8 19 9 9
```

Listing 2: Example of a jobscript file.

It should be noted that tools may require differently compiled versions of the binary file. Tools that require the same application binary settings can run within one job, while their respective estimated runtime must be taken into account. Measurements which require a different build of the application will run in a different job, which can contain the build instructions. To ensure comparability, when profiling the same application on different nodes, the nodes should have the same configuration. Computing nodes which are labelled with the same partition name have mostly same specifications. Otherwise, the user can select to run all measurements on exactly the same node, which prevents

parallel executions of the jobs, but guarantees accurate results. The jobs, though, can be submitted altogether at the same time and will remain in the queue until the node becomes free.

Aspects related to the execution of a measurement with respect to the runtime environment are also included in the script. These aspects are environment variables, a *cp* path, a *source* path, as well as additional modifications of the programs start command. Since it is assumed that a program is parallelised with an implementation of the Message Passing Interface (MPI), it is possible to specify modifications of the MPI commands as well.

A cluster-side directory structure is automatically created for the local package. This contains a database, the tool installations by the package manager and is also used for collecting the program outputs and the archived jobscripts.

Database. The database is used to store the status of the respective job, paths to the scripts and to the programs to be examined. Each jobscript changes the status of its associated program. These status changes make it possible to select which job is to be processed next after one finishes.

One advantage of using the database is that the progress of the job can be displayed to the user. Second, jobs that have been entered but not started yet can be cancelled or reordered. Also, optimisations can be made if, for example, two settings require the same compilation of the target program, so it is not carried out again.

To start jobs, a watcher script is currently used that queries the database at regular, user-definable intervals and starts new jobs. This has the advantage that the user does not have to maintain an active internet connection as soon as all Makefiles and jobscripts have been generated and the paths are stored in the database.

Output. The output of the application can be handled and post-processed in different ways. The basic option is to download the output file, but many tools offer their own browsers to visually display the output data, which will be covered as well. The visualised results can then also be downloaded.

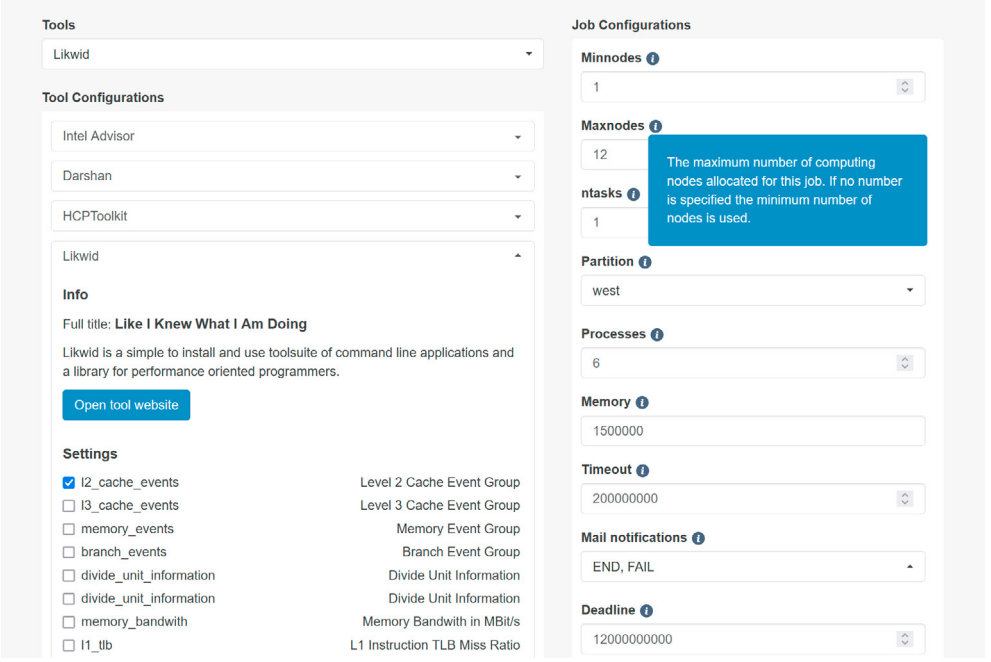

Fig. 4: A screenshot of the webapp. On the left side the tools and their settings can be selected. On the right side the job parameters can be set.

Utilisation. Figure 4 presents the configuration menu of a tool. There will be default settings for every tool with respect to the underlying hardware so that no user input is required for a sample run. Additionally, every setting has min. and max. values and hints for usage.

The abstracted underlying steps in the framework are the following:

- Installation of the tool with all the dependencies and configurations required for the setting
- Creation of a jobscript and a Makefile
- Path references to the Makefile and the jobscript are loaded into a database
- The watcher script checks the running jobs and adds runtime metrics to the database
- The watcher starts new jobs if resources are free by recompiling the programs with the information from the database and starting the jobscripts
- Inside jobscripts, start and end trigger scripts convert the status in the database when a program run is complete
- Output files can now be downloaded, or the output will be post-processed if specified, and the visualised or analysed output can be downloaded

4.1. Implementation

The entire framework is developed in Python as a PIP package. The user interface is implemented as a web application using Flask, which can be conveniently accessed through any web browser and is easily replaceable or extensible. The framework can either be executed locally by the user or deployed on a webserver. All data and commands are exchanged with the HPC cluster via an SSH connection, since most clusters have no alternative open port.

The installation of all tools and their dependencies is handled by the package manager Spack. Spack offers the advantages of being able to keep several versions of a package installed at the same time, automatically installing all necessary dependencies of a package, and that each installation process can be expressed in a uniform syntax. Especially the latter allows a uniform configuration for the existing tools and facilitates the integration of new tools into the framework. The configuration files for the analysis tools map the format from Figure 3 into YAML syntax, and these files alone can be used to conveniently add more tools to the framework.

On the cluster side, a PostgreSQL database is used to manage the created jobs and associate them with their modified Makefiles and generated jobscripts. This way the user does not have to rely on an active connection to the cluster and the database provides a simple interface to retrieve information about the jobs. To run the parallelised programs with the tools on the cluster, the framework relies on the job scheduler SLURM. To send pending jobs from the database to SLURM as well as terminating frozen jobs in SLURM, there are regularly calls to a script via the cron daemon.

5. Conclusion and Future Work

In this paper, the problems faced by scientific researchers in developing their software for massively parallel systems are illustrated. The need for a user-friendly framework for performance analysis is justified and a solution, which not only features a unified graphical interface, but also enables the automation of measurements and hides all technical obstacles, is presented. Furthermore, an approach for abstract and extensible representation of analysis tools that can be processed by the framework is provided. This can save scientists a lot of time and effort, and thus expenses, which they can invest in their own research instead. The entire framework is publicly available in a GitHub repository[3].

The framework supports performance evaluation and optimisation by facilitating access to and use of appropriate tool software. The following suggestions for further automation, abstraction and autonomisation can improve the evaluation of software, its optimisation and thus the use of resources in the HPC domain. The computer science aspect itself can also be further minimised for the end user, allowing domain experts to focus on their scientific research.

5.1. Improvements on the Framework

There are several components that need further improvement and development in the framework.

The job scheduling described in the previous section needs to be proven and improved for optimal resource utilisation. Since tracing and profiling for a single tool can take hours (depending on the application and data), parallel job execution is mandatory. At the same time, it will be tried to introduce a runtime estimation to help the user know

[3] https://github.com/wr-hamburg/hpcToolsFramework-client

when to expect the results. This step is far from obvious, as there is no magic formula and resulting time increases are usually not linear. The user could provide an estimate for pure application runtime to help the framework. Additionally, the user will be advised to choose appropriate input parameters to keep the runtime as short as possible, but as extensive as necessary to see the effects being studied.

It is not necessary to run all tools to obtain a good overview of the runtime and potential bottlenecks. However, knowing which tools (and how they are configured) can find 80% of performance losses in 20% of analysis time is not trivial. It is planed to introduce an intelligent analysis method with modes like “brief overview”, “easiest to fix” and “extensive analysis” as an implicit testing strategy for when the user does not specify a particular toolset. This can be a static approach with a series of simple runs or a dynamic approach where the results are evaluated in between to decide if another run would be helpful.

5.2. *Global Workflow*

Automated performance analysis is a big gain in HPC fields. It is anticipated to introduce a global workflow into the framework with the aim to implement a generic end-to-end optimisation procedure. First of all, the investigation steps are performed in a structured sequence as shown in Figure 5. Based on the results of the investigations, the framework can recursively select tools and start new investigations until performance problems have been broken down to the cause.

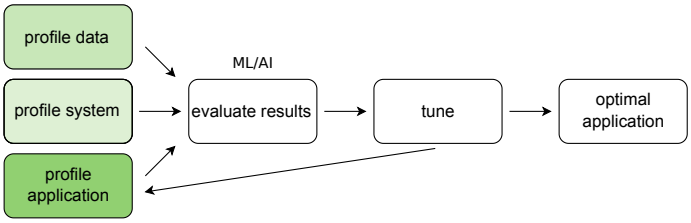

Fig. 5: End-to-end workflow towards an optimal application run. Every step is aimed to run automatically. Green modules are currently work in progress – the darker the colour, the further in progress.

Benchmarking: Domain experts often struggle to evaluate their performance results correctly. It is then essential to understand the limitations of the running environment – hardware, software and their interaction, which is very challenging. Therefore, users may not recognise performance losses, may perform their optimisation estimations guided or have to contact external staff. There is the plan of adding an optional system benchmarking step, where first a collection of static metrics of a system takes place, followed by a calculation of the peak performance of each component. This is then recalculated for the number of nodes most recently involved. Thus, these theoretical values indicate the maximum possible performance. This is lowered by the interaction of different layers, so dynamic benchmark runs are required to better estimate the actual values which are aimed to be achieved.

Nevertheless, benchmarking the system is a very resource intensive step and will be introduced very carefully by means of warnings before use. By default, it is planned to benchmark the minimal path (1-to-1 node interaction) and refer to further possible interactions. The footprint of the system is saved to a file and attempted to be accessed before the run. The existing benchmarks can be categorised and integrated into the GUI and scheduling system [18].

Based on the results, the application performance will be compared and can be assessed, which is a huge gain on the way to optimal resource usage. This evaluation step should involve machine learning and artificial intelligence techniques, since simple comparison of values does not meet to the challenge.

Generic Autotuning: Once the bottlenecks have been detected *and* ranked in the application, the user would need to optimise them manually. Autotuning is the cherry on top of the workflow, though there is a lack of suitable tools. As for example approchaches like [7] give optimisation hints to the user but the actual application and translation into code is still left to the user.

The applicability of machine learning techniques will be discussed in the following:

The outputs of the tools serve as input for the ML models, which are mapped to the previously identified performance bottlenecks. This mapping can be learned and generalised by an ML architecture.

Instead of concrete suboptimal-performance causes, user behaviour can be analysed. For certain tool results, it is possible to identify what follow-up studies have been done with which tools and with what degree of success. It could be concluded that the same investigations will help with a similar observed problem.

These approaches are shaping the path towards self-optimising software and autotuning. Once the causes of performance problems have been identified through a tool and ML use, they are mapped back to solutions in the form of code semantics. Putting all aspects together will enable automatic optimisation of input programs.

In contrast to the state-of-the-art autotuning approaches, not only application parameters, compilers, highly specialised mathematical calculations or boundedness characteristics (IO, CPU, memory) are considered with the help of ML and direct analysis of the hardware performance counters [5]. Rather, the abstract tool-based analysis in combination with human-made semantical optimisation may allow a direct mapping of performance weaknesses to program semantics. The whole can be extended to energy and cost efficiency goals.

Feedback Module and Hints: Whenever an automatic evaluation or decision could not be performed successfully, the user receives a detailed and extensive feedback with a description of all problems. In between, it is possible to add additional hints or decisions for each step and repeat each step separately. This allows for a fine-grained analysis that makes the most of the user's potential knowledge of the application.

Data Analysis: Hardly any tool takes the data content into account. When large simulations write terabytes of data, compression becomes necessary to save storage space or even relieve the network, if less data would be transferred [9]. Once the data has been detected to be well compressible, like is done automatically in [4], code insertion tools can make use of features of widely used middleware libraries in HPC (HDF5 and NetCDF filters).

References

[1] Agrawal, Ankit, and Alok N. Choudhary (2016) "Perspective: Materials informatics and big data: Realization of the "fourth paradigm" of science in materials science" *APL Materials* **4**: 053208.

[2] Klôh, Vinícius, Matheus Gritz, Bruno Schulze, and Mariza Ferro (2019) "Towards an Autonomous Framework for HPC Optimization: Using Machine Learning for Energy and Performance Modeling" *Anais do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*.

[3] Madsen, Jonathan R., Muaaz G. Awan, Hugo Brunie, Jack Deslippe, Rahul Gayatri, Leonid Oliker, Yunsong Wang, Charlene Yang, and Samuel Williams (2020) "Timemory: Modular Performance Analysis for HPC" in Springer International Publishing *High Performance Computing*: 434–452. isbn: 978-3-030-50743-5

[4] Plehn, Julius, Anna Fuchs, Michael Kuhn, Jakob Lüttgau, and Thomas Ludwig (2022) "Data-aware compression for HPC using machine learning" *Data-aware compression for HPC using machine learning* : 8–15.

[5] Benkner, Siegfried, Franz Franchetti, Hans Michael Gerndt, and Jeffrey K. Hollingsworth (2014) "Report from Dagstuhl Seminar 13401 Automatic Application Tuning for HPC Architectures".

[6] Orr, C. H., R. R. Mcfadden, C. A. Manduca, and L. A. Kempler (2016) "Resources and Approaches for Teaching Quantitative and Computational Skills in the Geosciences and Allied Fields".

[7] Zhou, Keren, Xiaozhu Meng, Ryuichi Sai, Dejan Grubisic, and John Mellor-Crummey (2022) "An Automated Tool for Analysis and Tuning of GPU-Accelerated Code in HPC Applications" *IEEE Transactions on Parallel and Distributed Systems* **33**: 854–865.

[8] Knobloch, Michael, and Bernd Mohr (2020) "Tools for GPU Computing - Debugging and Performance Analysis of Heterogenous HPC Applications" **7**: 91-111.

[9] Kuhn, M., Julian Kunkel, and Thomas Ludwig (2016) "Data Compression for Climate Data" *Supercomputing Frontiers and Innovations* **3**: 75–94.

[10] Martínez, Víctor, Fabrice Dupros, Márcio Castro, and Philippe Navaux (2017) "Performance Improvement of Stencil Computations for Multi-core Architectures based on Machine Learning" *Procedia Computer Science* **108**: 305–314.

[11] Wu, Xingfu, Aniruddha Marathe, Siddhartha Jana, Ondrej Vysocky, Jophin John, Andrea Bartolini, Lubomír Říha, Michael Gerndt, Valerie Taylor, and Sridutt Bhalachandra (2020) "Toward an End-to-End Auto-tuning Framework in HPC PowerStack": 473–483.

[12] Johanson, Arne, and Wilhelm Hasselbring (2018) "Software Engineering for Computational Science: Past, Present, Future" *Computing in Science Engineering* **20**: 90–109.

[13] SchedMD (2022) "Slurm Workload Manager Documentation version 21.08". url: https://slurm.schedmd.com/documentation.html

[14] Killcoyne, Sarah, and John Boyle (2009) "Managing Chaos: Lessons Learned Developing Software in the Life Sciences" *Computing in Science Engineering* **11**: 20–29.

[15] Thompson, Neil (2017) "The Economic Impact of Moore's Law: Evidence from When it Faltered" *SSRN Electronic Journal*.

[16] Prabhu, Prakash, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August (2011) "A Survey of the Practice of Computational Science" in Association for Computing Machinery *State of the Practice Reports*. isbn: 9781450311397

[17] Virtual Institute - High Productivity Supercomputing (2022) "Tools Overview": url: https://www.vi-hps.org/tools/tools.html

[18] Welch, Aaron, Oscar Hernandez, and Barbara Chapman (2021) "Combining Static and Dynamic Analysis to Query Characteristics of HPC Applications" *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* 420–429.