A picture is worth a thousand numbers – Enhancing Cube's analysis capabilities with plugins

Michael Knobloch, Pavel Saviankou, Marc Schlütter, Anke Visser, and Bernd Mohr

Abstract In the last couple of years, supercomputers became increasingly large and more and more complex. Performance analysis tools need to adapt to the system complexity in order to be used effectively at large scale. Thus, we introduced a plugin infrastructure in Cube 4, the performance report explorer for Score-P and Scalasca, which allows to extend Cube's analysis features without modifying the source code of the GUI. In this paper we describe the Cube plugin infrastructure and show how it makes Cube a more versatile and powerful tool. We present different plugins provided by JSC that extend and enhance the CubeGUI's analysis capabilities. These add new types of system-tree visualizations, help create reasonable filter files for Score-P and visualize simple OTF2 trace files. We also present a plugin which provides a high-level overview of the efficiency of the application and its kernels. We further discuss context-free plugins, which are used to integrate command-line Cube algebra utilities, like cube_diff and similar commands, in the GUI.

1 Introduction

Cube is the performance report explorer for Score-P [14] and Scalasca [10]. The CUBE data model is a three-dimensional performance space consisting of the dimensions (i) performance metric, (ii) call-path, and (iii) system location. Each dimension is represented as a tree and shown in three coupled tree browsers, i.e. upon selection of a tree item the trees to the right are updated. Non-leaf nodes of each tree can be collapsed or expanded to achieve the desired level of granularity. Figure 1 shows an overview of the Cube GUI.

M. Knobloch · P. Saviankou · M. Schlütter · A. Visser · B. Mohr
Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany
e-mail: {m.knobloch,p.saviankou,m.schluetter,a.visser,b.mohr}@fz-juelich.de

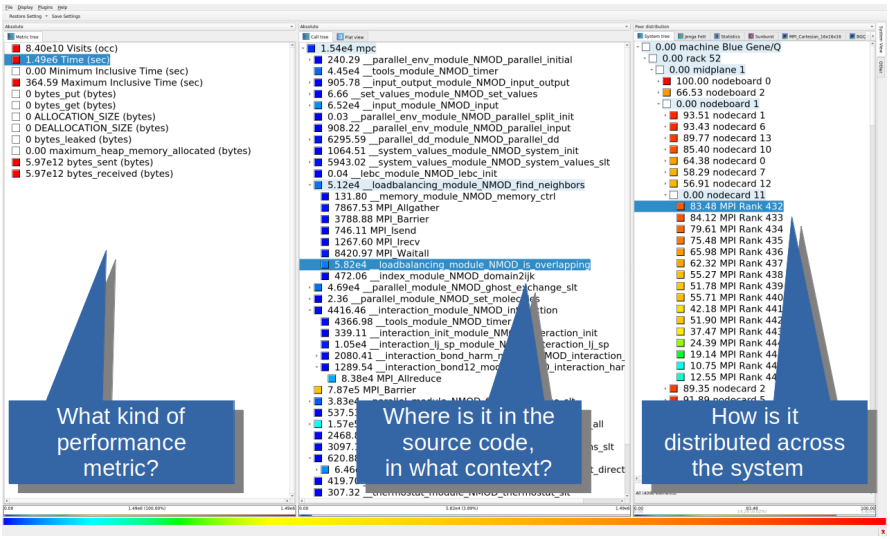


Fig. 1: Overview of the Cube GUI. It shows the three coupled tree browsers with the metric-tree on the left, the call-tree in the middle and the system-tree on the right.

Cube can be used to analyze measurements of all scales, from a laptop to the largest-scale supercomputers with millions of threads. It is regularly used in the Jülich Supercomputing Centre (JSC) Extreme Scaling Workshops [16] and for the analysis of applications in the High-Q club [7]. These large-scale applications typically embody an extensive call-tree and a system-tree with thousands of locations. Without further visualization, an analysis of such large trees is confusing and inefficient.

In recent years, supercomputing systems became more and more complex, both on the hardware and the software side. Many HPC systems now have heterogeneous nodes with some form of accelerator attached to the compute nodes. This leads to a higher variability in programing models used for HPC application development. In addition to the traditionally used MPI and OpenMP, we now have CUDA and OpenACC for GPU programming and OpenCL for FPGA's. All these programming models require new way of representation in the trees and new analysis methods.

To meet the challenges outlined above, we work continuously on Cube to enhance its analysis capabilities and to make it more scalable. With version 4, Cube evolved from being a simple report explorer to a capable analysis tool [18]. However, with the monolithic approach, which we followed in Cube for a long time, this is a daunting task as the code quality degrades and becomes harder to maintain. To counteract this, we split up Cube in multiple components and distribute it in form of four separate packages:

- CubeW – A high performance C library to write CUBE files.

- CubeLib – A general purpose C++ library to interact with and manipulate CUBE files and a set of associated command-line tools.
- CubeGUI– The graphical report explorer.
- jCubeR – A Java library for reading CUBE files.

To add advanced analysis features more easily, we introduced a plugin infrastructure to the CubeGUI. We separated core parts, that build the foundation of the GUI, and reformulated the other parts of the CubeGUI as standalone plugins which are shipped together with the core parts in the CubeGUI package. Plugins, which may have further dependencies or are not considered stable, are available for download at our website [3]. Other performance analysis tools follow this route as well, for example TAU provides a plugin infrastructure as well [15].

The remainder of this paper is organized as following: In section 2 we describe the plugin architecture in more detail. Section 3 covers context-free plugins and in section 4 we present plugins that help with the analysis of large-scale applications by enhancing or replacing the system-tree view. Several other plugins, their use-case and examples are presented in section 5. These include stable plugins, that are included in the CubeGUI package and more experimental plugins that are available in an online repository. We conclude the paper and give an outlook on future work in section 6.

2 Plugin architecture

The CubeGUI plugin architecture is designed to further advance the extension of the analysis features of CubeGUI, while at the same time streamlining the extension process. The first step towards using a plugin infrastructure and the decoupling of features consisted in defining core functionality and extensions. In this context some features of the previously monolithic CubeGUI have been classified as extensions and turned into plugins, even though they always have been part of the CubeGUI.

The core functionality of the CubeGUI consists of Cube file management, GUI handling and the global calculation mode. The management aspects cover loading the Cube data and meta data for metric-, call- and system-tree descriptions. The core elements of the CubeGUI structure are three coupled tree browsers, that can be seen in Figure 1. Using this three dimensional approach, the calculations happen in a three step selection process from left to right, each step narrowing the focus of the calculation. The name Cube is derived from this three dimensional approach. For the selections single or multiple entries can be chosen, although the metric-tree only allows selections of the same type, e.g., counts, time or bytes. In this scheme, the right-most panel represents a point-like value depending on the selections on the two left panels. The middle panel aggregates along a row-value depending only on the left-most panel. The left-most panel is an aggregation on a plane value and has no dependencies, representing the global value. The default setting has the metric-tree on the left, the call-tree in the middle and the system-tree on the right. However, the order of panes can be changed, with the respective shift in the calculation order.

Extensions for the CubeGUI are the foundation of the plugin concept and can fall into a set of different categories.

The core behavior of CubeGUI assumes to be working on a single Cube file. For a new class of plugins this is not a requirement, as they work on one or multiple existing Cube files and create a resulting Cube file in the process. These so-called *context-free plugins* will be covered in Section 3.

Cases where more than a single number is used for entries are implemented by *value plugins*, which change the handling and display of values in the CubeGUI. These can for example occur in forms of histograms, small box plots, or a numeric triple. An application for this plugin is the visualization of TAU [19] profiles in the CubeGUI, see Section 5.

Aside from numerical values, the CubeGUI represents values through colored nodes, taken from a global color map. The color mapping allows easy visual identification of hot-spots and patterns. While the default color map can be configured to a degree, there are occasions where a more specialized color map is required. Whether this is a device optimized color map or map highlighting a specific use case, in either case a new *colorMap plugin* can be employed.

In Section 4 another category of extensions is presented. There, additional and alternative visualizations for the default system-tree are presented, with a special focus on a global perspective. These fall into the category of *context-sensitive plugins*, as their value representation is dependent on the selections in the first two panes.

All extensions have the commonality, that they require the use of an API to define the plugin and interact with the CubeGUI core. This API is part of the overall plugin architecture and the interface between core and current and future plugins. It realizes a set of states that can be queried and signals that plugins can react to. For more detailed information refer to the set of examples in the CubeGUI installation and the documentation, particularly the CubeGUI Plugin Developer Guide[1], which is included in a standard CubeGUI installation.

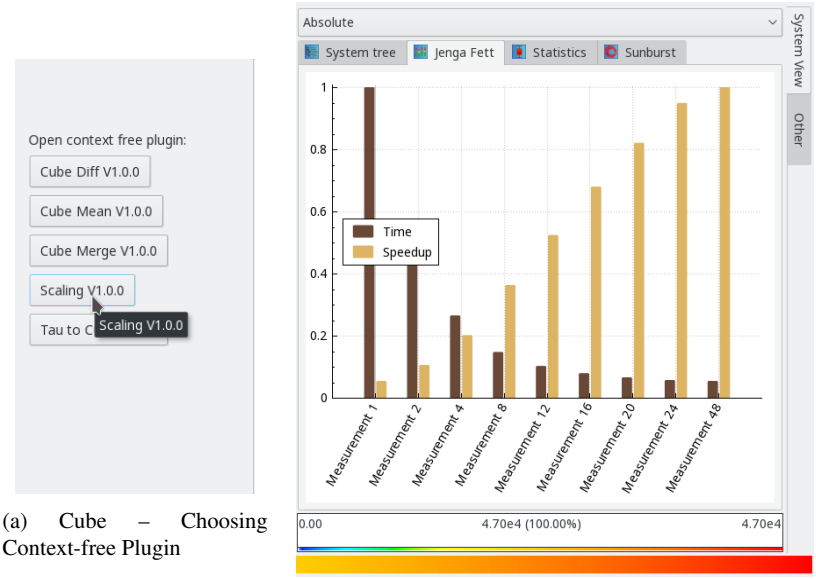
In the following we present examples for the different categories of extensions.

3 Context-free plugins

As stated in section 2, context-free plugins are a special kind of plugins that are only available when no Cube files are loaded. They enhance the loading screen of the CubeGUI to integrate operations that generate Cube files within the GUI. Upon start, the CubeGUI shows a list of available context-free plugins next to a list of recently opened Cube files, see Figure 2a.

One main purpose of these plugins is to provide access to the Cube algebra utilities (which are part of the CubeLib package) directly from the GUI. These are:

[1] Plugin development guide: `https://apps.fz-juelich.de/scalasca/releases/cube/4.4/docs/plugins-guide/html`



(a) Cube – Choosing Context-free Plugin

(b) Cube Scaling

Fig. 2: Screenshots of Cube showing the loading screen where the context-free plugins can be chosen (a) and an example of the *Scaling* plugin in combination with the *Jenga Fett* plugin (b) There we plot the time in dark brown and the corresponding speedup for measurements from 1 to 48 processes.

- *cube_merge* allows to merge several experiments into a single one and explore the result. It is typically used for an analysis that requires more metrics than can be collected in a single measurement, e.g. hardware counter measurements with PAPI or perf counters, where only a limited number of counters can be measured simultaneously. It can also be used to enhance a Scalasca trace analysis report (which omits any hardware counter information that might be present in the trace) with hardware counter information obtained from a profile report. This is necessary for a detailed POP analysis with the *Advisor* plugin as presented in section 5. Further, *cube_merge* is useful for a comprehensive analysis of MPMD applications, where each part was measured independently, or workflows consisting of multiple executables.
- *cube_mean* creates an ”average” result ouf of several structurally identical measurements in order to smooth the variations in the run-time, introduced for example by OS jitter or contention on the network.
- *cube_diff* creates the difference between two preferably structurally identical measurements. The typical usage example is the validation of tuning actions with a ”before optimization vs. after optimization” comparison. It can also be used to investigate the behavior of an application built with different toolchains, e.g. compiler or MPI versions.

Beside the Cube algebra tools, context-free plugins can be used to integrate other tools as well. We provide two additional context-free plugins:

- *tau2cube* enables the CubeGUI to load native TAU performance measurements. We will discuss the *tau2cube* plugin in more detail in section 5.
- *Scaling*: Investigating the scaling behavior of an application is a common task for an HPC application developer. Either the application is run with the same input on different scales (strong scaling) or the input set scales with the number of system resources (weak scaling). However, usually only the whole application or a few kernels are regarded in such an analysis.
  The *Scaling* plugin allows a detailed analysis of the scaling behavior of every single routine. For that, a series of "identical" Score-P measurements on different system sizes is performed. This results in cube files with the same metrics and a nearly identical call tree. Only the system tree is different in each of these files. The *Scaling* plugin gathers the individual measurement results into one and displays the metric values in dependency of the system size, e.g. time/#processes. It is recommended to use the *Scaling* plugin in combination with the *JengaFett* plugin (see section 5) to replace the system tree view. Figure 2b shows an example of the *Scaling* plugin where the system dimension displays bar plots for the time per process (dark brown) and the calculated speedup (light brown). This calculation works for every metric and call-path selection.

4 System-tree enhancements

The system-tree view is the default right-most pane of the CubeGUI representing the system locations, e.g., processes, threads, CUDA streams etc., used in the measurements. It combines a logical hierarchy of processes and their child threads with known hierarchal information about the system hardware up to the rack level (e.g. nodes, midplanes, etc.). Each level can be collapsed and expanded as needed and the respective levels will show the inclusive or exclusive values, as with the metric- and call-trees. It also provides the option to define and select subsets that may be used for example by the box plot plugin as shown in section 4.2.

Figure 3 shows a measurement of MP2C on 4096 processes on a Blue Gene/Q machine. MP2C [22] is a simulation for multi particle collision dynamics of solvated particles in a fluid. The the system-tree shows the hierarchy levels of the Blue Gene/Q from rack to node card. Since MP2C is a MPI only application, the node level hierarchy is limited to processes. This example will be used to showcase the visualization alternatives presented in this section.

The reason for introducing alternative system-tree visualizations lies in the scope and variation of applications and users' analysis needs, where one solution rarely meets all requirements. Most of the time a combination of different view points on the same data, is the most helpful approach. Therefore, the intent of this category of plugins is not to replace the system-tree, and instead offer a set of different perspectives to be used in unison. With that in mind, the plugins are not completely

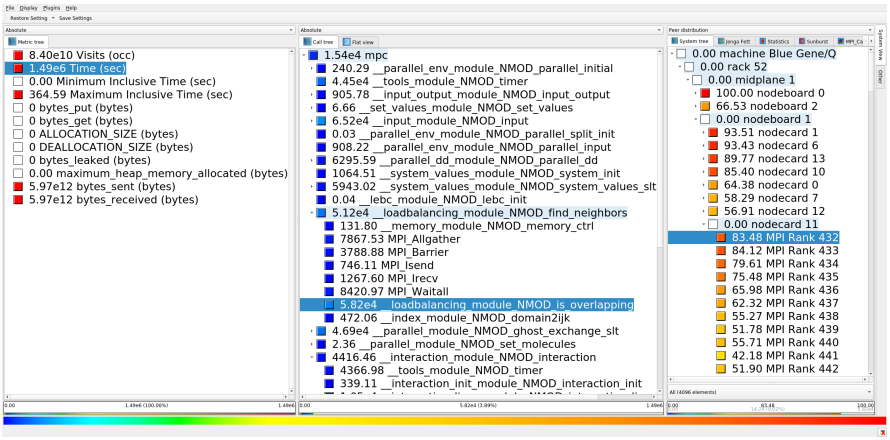


Fig. 3: MP2C measurement example with 4096 processes on a Blue Gene/Q highlighting the limited global overview with the default system-tree due to the limited number of locations shown at the same time.

disconnected in their function and allow selections in one view to update selections in others where applicable. With the ability to detach views, these can be viewed side by side.

Any measurement containing notably more locations than the standard view size of the system-tree in the CubeGUI presents a challenge for its comfortable use. In Figure 3 the number of processes able to be shown at the same time is limited and even collapsing the tree to node card level will not improve that notably. In consequence, the user has to spend time scrolling to specific locations and looses the global view over all locations. In turn, this may lead to missing patterns in the behavior of the current metric when spanning multiple locations. To alleviate this issue, various Cube plugins have been introduced to enhance the system-tree view or to offer alternative visualizations.

The following section highlights alternatives that use numerical or visual presentations and, in case of topologies, incorporate additional data to create the visualization.

4.1 Sunburst

This plugin displays the system-tree data in a 360 degree sectored disk [20]. It allows the visualization of the whole system-tree in a relatively limited space which provides a global overview over the value distribution of a metric over the entire measurement.

The system-tree hierarchy is reflected in the rings of the sunburst view, from the highest level at the center to the location level on the outermost ring. Any selec-

tion of locations or levels is mirrored in the system-tree panel and vice versa. The sunburst view can be manipulated to show different levels at the same time through expanding and collapsing different selections. Settings for behavior and visual style can be accessed through a context menu.

In continuation of the initial example of Figure 3, Figure 4 shows the same metric and call-path selection to highlight the differences. While in the the system-tree none of the location sub sets reveal a pattern directly, in the sunburst view the regular pattern becomes immediately recognizable. While the sunburst view does not show numerical data directly, except through mouse-over on slices of the respective levels, it aids in the identification of locations that justify further investigation. Making selections here will update the system-tree view accordingly and will allow a closer look.

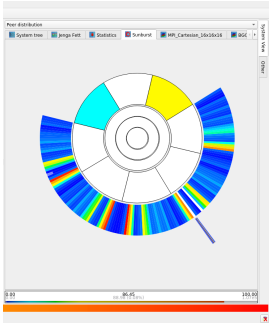


Fig. 4: MP2C example with an equivalent sunburst view of Figure 3.

4.2 Statistics: Box and Violin Plots

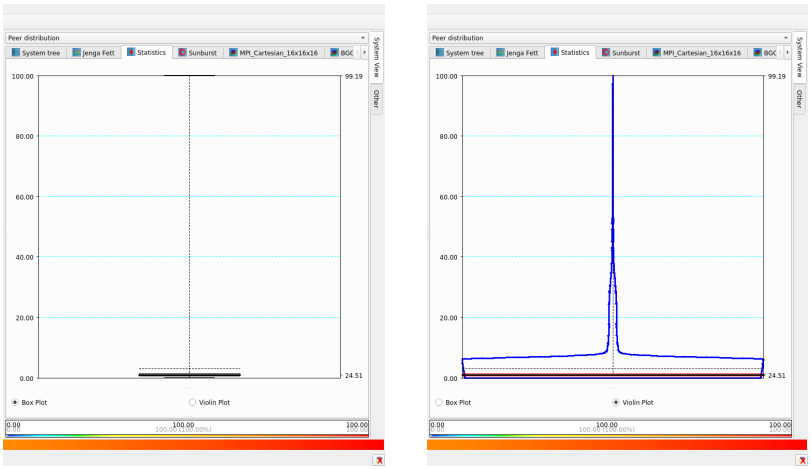
The *Statistics* plugin gathers data for the selected metric values from the system-tree and displays these in form of a box plot [9]. The statistical data is represented as maximum and minimum, quartiles, as well as mean and median.

This plugin allows fast access to global numerical data of a measurement and can serve as entry point by offering a global perspective as well as highlighting more detailed imbalances, depending on the chosen metric and call-path selections.

For a detailed analysis, there also exists the option to define subsets of nodes, processes or locations and limit the statistics calculation to that subset.

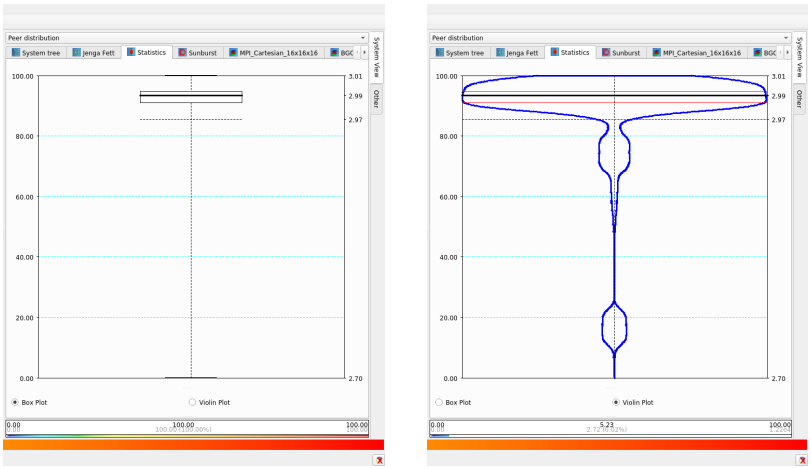
The violin plot [11] is similar to the box plot, however it additionally presents the distribution of the metric values. This allows the identification of partitions within the system and highlights peaks in multi-modal measurements, which are an indication for the existence of performance issues within the measurement. In combination with the Sunburst of section 4.1 or the topology plugin of section 4.3 this aids in matching the general distribution or specific partitions to sets of locations. The statistics plugin also offers the numbers in tabular form for easy access.

Figure 5 uses the MP2C example from before, and shows a concentrated partition with only a few outliers, which is the expected result based on the visual impression of the sunburst in Figure 4. In this case the shape of the violin plot does not reveal too much additional information. Compared to this, the example of Figure 6, looking at a different region of the same experiment, highlights multiple partitions for the respective selection. As mentioned before, not all visualizations provide the most insight for every use case and therefore they should be used together to reveal the most significant issues.



(a) Box plot (b) Violin plot

Fig. 5: MP2C example showing a statistics perspective of the selections and data from Figure 3.



(a) Box plot (b) Violin plot

Fig. 6: Different function selection on the MP2C example, highlighting a distribution with partitioned clusters.

4.3 Topologies

The topology plugin, compared to the sunburst and statistics plugin, is not just a different view on the same system-tree data as it incorporates additional informa-

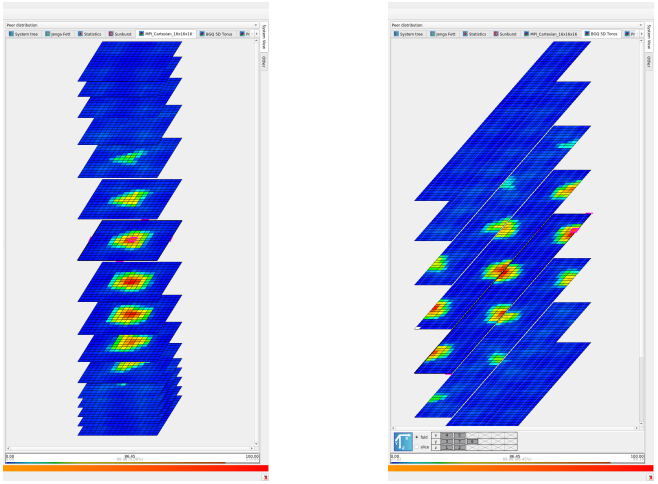
tion about neighborhood relations between locations. This additional structural data can be defined by MPI Cartesian calls, Score-P user instrumentation or platform specific interfaces, e.g., the 5D torus of a Blue Gene/Q like the now decommissioned JUQUEEN [21] in Figure 7b. More details on the creation and recording of topologies can be found in the user documentation on the Score-P website [4].

The main component of the topology plugin is the 3D display of the topology data. It allows the visualization of up to three dimensions directly, while for higher dimensional data dimensions have to be combined through either folding or slicing.

Folding in this context represents the concatenation of two or more dimensions into one visual dimension. By Slicing on the other hand, the user chooses single values for a subset of dimensions and displays the remaining dimensions in the plugin. Both methods reduce the number of visual dimensions effectively to three and allow the visualization of topologies with an arbitrary number of dimensions. The arrangement and order of dimensions can be controlled by the pull-up control field at the bottom of the plugin. Controls for a more fine grained adjustment of the 3D view can be found in the topology toolbar. Additional settings for the visual appearance of the 3D view are available in the Plugins menu. As with the sunburst plugin mouse-over reveals the numerical information to the visual data.

The sunburst view showed a repeating pattern for the selected call-path in the MP2C example. Figure 7a displays the used MPI topology, which was automatically created from the MPI data by Score-P. This visualization now reveals that the repeating pattern represents a hot-spot in the communication pattern, as the view incorporates the Cartesian structure provided by MPI. In Figure 7b the same selection is presented in the platform topology of JUQUEEN, showing the seven dimensions of the Blue Gene/Q architecture. This shows the physical placement of processes and threads within the 5D torus and the assignment to cores and hardware threads within their nodes. In this figure the example uses folding to arrange the dimensions. With high dimensional topologies this may require checking various dimension orders, but as the example shows hot spots can be identified as with the MPI topology. The additional hardware information can lead to the identification of issues with the chosen partition within the system or of outside influences that are not caused by the application itself. As this combines logical with physical locations, threads have to be bound to cores for the duration of the application run, otherwise an unambiguous matching is impossible.

Aside from a straightforward *Process x Threads* topology, which is a two dimensional mapping of the system-tree and can be generated for every measurement, all other topology types require additional input from the application, the user or the system under investigation. That makes them a conditional tool to be employed if the use case matches the requirements. This showcases that the topology plugin, like the other plugins presented in this section, should and have to be used in concert with each other and that there is no hard rule for when one plugin should be preferred over the others.



(a) MPI Cartesian 16x16x16 (b) Blue Gene/Q 5D torus

Fig. 7: Topologies for the MP2C example of Figure 3.

5 Other Plug-ins

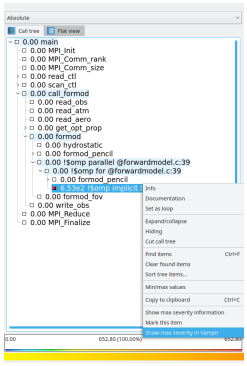
All plugins presented so far fit in Cube's 3-dimensional data model with a metric, a program and a system dimension. However, the plugin architecture is not limited to this scheme. In this section we present plugins that still work on the selected metric and call-tree item, but either open a new window or show their results in the rightmost panel, but independent from the system-tree. For that we extended the right-most panel by a tabs to switch between the system dimension and the other plugins.

Integration in the Score-P ecosystem

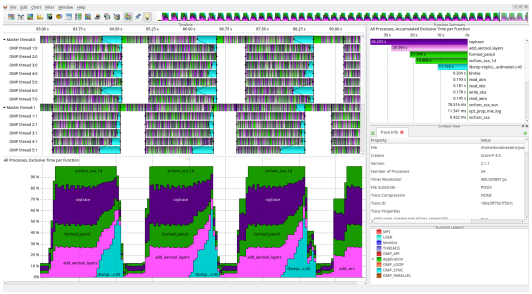
One of the major goals of the Score-P ecosystem is the interoperability of distinct performance analysis tools built upon the common measurement infrastructure. This is ensured by common data formats for profiles - the CUBE4 format - and traces - the Open Trace Format 2 (OTF2) [8]. OTF2 trace files can be analyzed manually by Vampir [13] or automatically by Scalasca [10].

Vampir connector

While it is already useful to be able to analyze the same trace data manually and automatically, it would be preferable to use the results of the Scalasca trace analysis for a following in-depth analysis with Vampir. Scalasca stores detailed information of the most severe instances, i.e. the instance with the longest waiting time, for each performance inefficiency pattern it detects. While this is unambiguously for point-to-point communication, it is defined as the instance with the largest *sum* of waiting times of all involved processes/threads for collective communication. This is not necessarily the one with the largest individual waiting time.



(a) Cube – Context menu to open Vampir



(b) Vampir zoomed to most severe instance

Fig. 8: Screenshots showing the *Vampir connector* plugin. The user can start Vampir directly from the CubeGUI (a), which opens the trace at the relevant point in time (b).

If Vampir and the D-BUS components are installed on the same machine it is possible to connect the CubeGUI to the trace browser and view the state of the analyzed program at the point of the occurrence of the most severe instance of the selected pattern. Figure 8 shows an example using the JURASSIC (Juelich Rapid Spectral Simulation Code) application [12]. The user has to select the desired metric in the metric-tree and then right-click on the respective instance of that pattern in the call-tree to open Vampir, as shown in Figure 8a. Here the selected metric is the ”Wait at OpenMP barrier” and the interesting instance is the implicit barrier at the end of the main loop. This in turn opens Vampir (in a separate window) at a reasonable zoom level so that the pattern and some application activity before and after is visible, see Figure 8b. We see the last three iterations of the computational loop with the typical increase in waiting time due to load-balancing issues.

tau2cube and *Tau Value display*

TAU [19], as part of the Score-P ecosystem, can open CUBE4 files natively. That does not hold for the opposite direction. We provide a (context-free) plugin called tau2cube to enable the CubeGUI to load native TAU measurements. It is possible to load and directly merge multiple TAU measurements, which is useful as TAU stores all recorded metrics in distinct files, with the exception of the time metric, which is present in each file. Figure 9 shows an example of a TAU measurement with four metrics. Note that we see a flat call-tree as TAU stores only flat profiles.

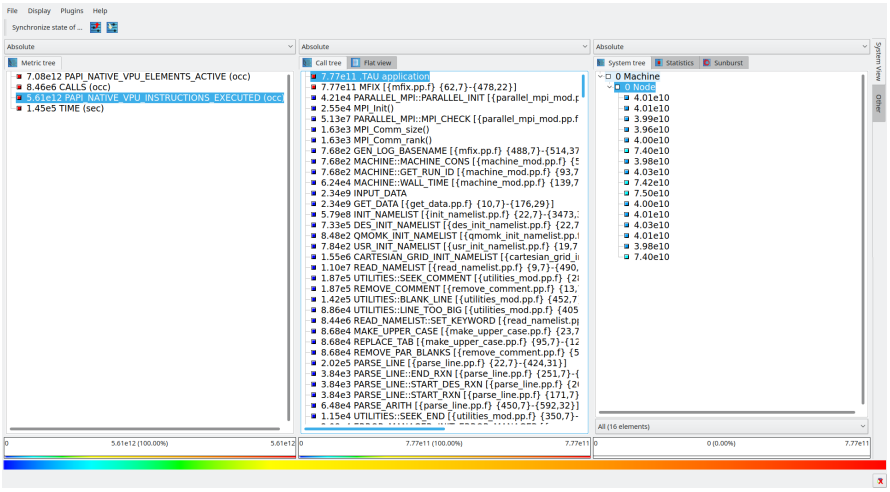


Fig. 9: Cube showing the merged result of 3 TAU measurements. We see a flat call-tree originating from TAU's flat profiles and no names of the nodes in the system-tree, as TAU does not store them.

Score-P is able to collect a statistic of metric values for every call-path. These are called "Tau Tuples" as they follow the same structure as the tuples introduced by TAU. It is possible to display them as a small box-plot in the metric- and call-tree. This allows to get an overview about statistic behavior along the call-tree or system-tree

ScorePion

With instrumentation being the default measurement mode of Score-P, measurement overhead is a factor to consider in many analyses, especially of C++ applications with many small functions that get called frequently. To mitigate that effect we can use filtering, i.e. mark functions to not be measured (run-time filtering) or not be instrumented at all (compile-time filtering). The GNU compiler uses the same

format for compile-time filtering as we use for run-time filtering in Score-P. The format of the filter file used by the Intel compiler[2] varies slightly.

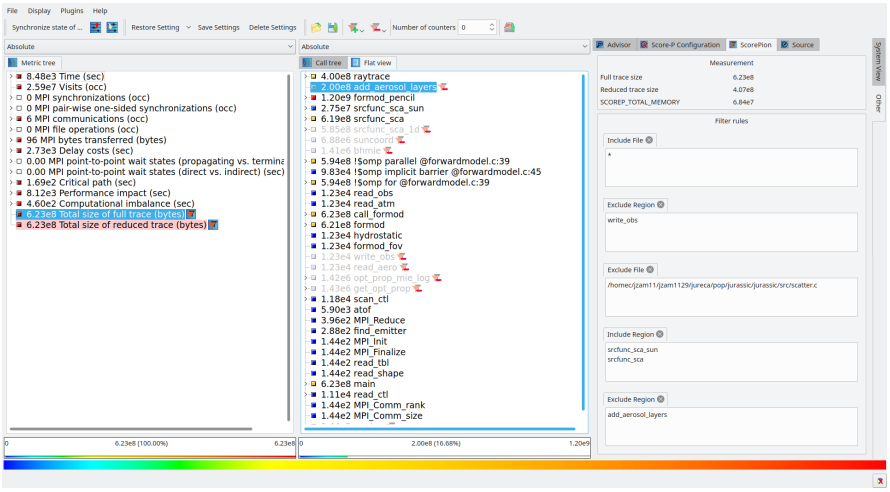


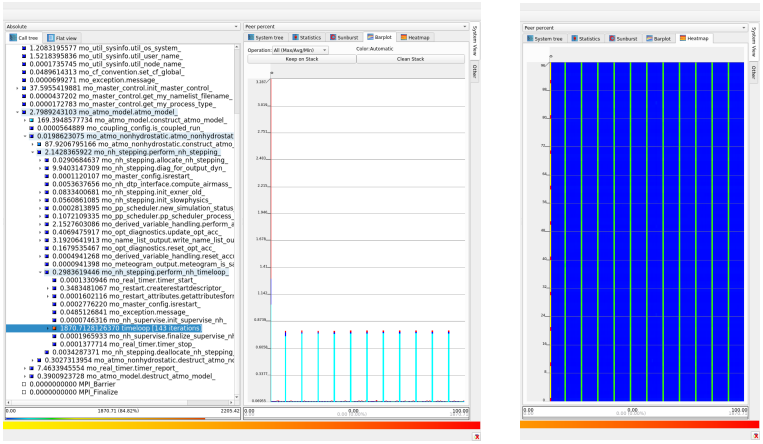
Fig. 10: Screenshot of the *ScorePion* plugin. In the right pane it shows the stacking of filter rules and information on trace size and memory requirements.

With the *ScorePion* plugin we enable the creation of a Score-P or Intel filter file directly from the GUI. The user simply right-clicks on a call-tree item to add or remove it from the filter. The *ScorePion* plugin generates additional metrics for Score-P measurement system memory requirements, the resulting trace size (after applying the filter), and the expected impact on measurement overhead, see Figure 10. All the advanced features of the Score-P filter file like black- and white-listing of functions and files, stacking of filter rules and wildcards are supported by *ScorePion*.

Iteration profiling

Score-P enables the user to mark loops via its user instrumentation API and record each iteration of the loop independently. The default representation of such a loop is a separate call-tree node for each iteration, which makes an analysis of loop-dependent behavior very difficult for loops with many iterations. We provide two plugins for a graphical analysis of iteration-dependent behavior. Figure 11 shows an example of both plugins with the TeaLeaf [5] application. It clearly shows that every 12th iteration shows a different behavior from the previous 11. This is due to a function that is called only every 12th iteration. The *Barplot* plugin plots the value of the

[2] Intel compile time filtering API description: `https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-tcollect-filter-qtcollect-filter`



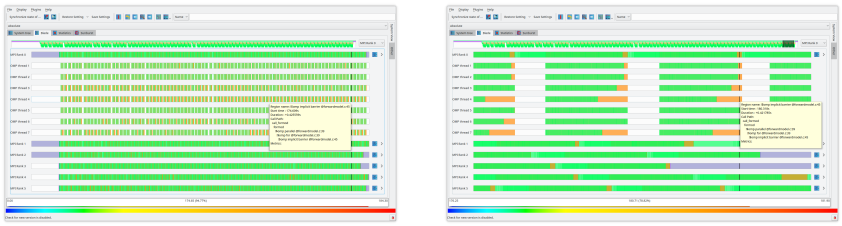
(a) Barplot (b) Heatmap

Fig. 11: Screenshots of an iteration profile of TeaLeaf using the *Barplot* (a) and the *Heatmap* (b) plugin, showing the Time metric in each case. Both views show an anomaly every 12th iteration.

selected metric vs. the iterations of the loop. The value can be the minimum, maximum or the average across all system locations. Further, a stacked bar of minimum, average and maximum is possible, as shown in Figure 11a. The *Heatmap* plugin (Figure 11b) plots locations vs. iterations with the value being color-coded according to the currently chosen color palette. The color palette can be changed using the *Colormap* plugin. Next to the standard rainbow palette, Cube provides a configurable gradient, double gradient, helix, and different standard gradient palettes. Using those, specific values such as the median or extrema can be emphasized. It also allows to adopt visualization for screenshots used in printing or presentations.

Blade

Scalasca's automatic trace analysis guarantees to cover the entire trace data, but the generated report omits the time dimension. However, often it is necessary to look at the dynamic behavior of the analyzed application with a timeline-based trace browser. The standard tool for this task in the Score-P ecosystem is Vampir, a very powerful OTF2 trace analyzer with many customizable displays. Vampir however is a commercial tool and typically only available on larger supercomputers. For small-scale experiments a quick glance on the trace is often enough to identify performance problems. For that we provide *Blade*, a simple OTF2 trace explorer, which is integrated in the CubeGUI. Thus, it allows a quick look on the tracing experiments with respect of the selected call-path and simple filter rules.



(a) Blade – Full trace view (b) Blade – zoomed to iteration level

Fig. 12: Screenshots of the *Blade* plugin for an execution of JURASSIC. User routines are colored green, MPI routines purple and time spend in OpenMP in orange.

Figure 12 shows screenshots of the *Blade* plugin. A view of the entire application execution is shown in Figure 12a. Figure 12b shows the same information in *Blade* as Figure 8b shows in Vampir, i.e. the iterations with the highest waiting time in the OpenMP barrier. Vampir shows a lot more information, but the general structure of the imbalance leading to the wait-state is also visible in *Blade*. However, the automatic zooming to the most severe instance is not (yet) available for *Blade*, so a complete manual analysis is required in this case.

Program structure

Raw performance data is often very hard to interpret without detailed understanding of the applications algorithm and implementation. We developed two plugins to help the performance analyst to assess the structure of the application by providing a complete *Call Graph* and its implementation by linking performance data to the source code. Figure 13 shows examples of both plugins.

Call graph

This plugins displays the call-tree in form of a graph. The unique regions are the nodes of the graph and aggregated metric values are assigned to the edges. A call-graph can help to detect critical calls in an application with a complex call-tree more efficiently. This plugin generates the call-graph in the dot format and thus depends on a Graphviz[3] installation. A new Window is opened containing the graph, as presented in Figure 13a.

[3] `http://www.graphviz.org`

Source Code Viewer

Source code viewer with syntax highlighting for C/C++ and Fortran. The viewer, like the system view, is linked to the call-tree, i.e. selecting a different call-tree node automatically shows the respective source code region. An example is shown in Figure 13b, highlighting the main OpenMP loop of the JURASSIC application.

Metric correlation

Often it is necessary to regard the combination of multiple metrics in order to get a complete picture of the application performance characteristics. To spare the user from clicking through the metric-tree we provide two plugins that help to identify correlation between metrics.

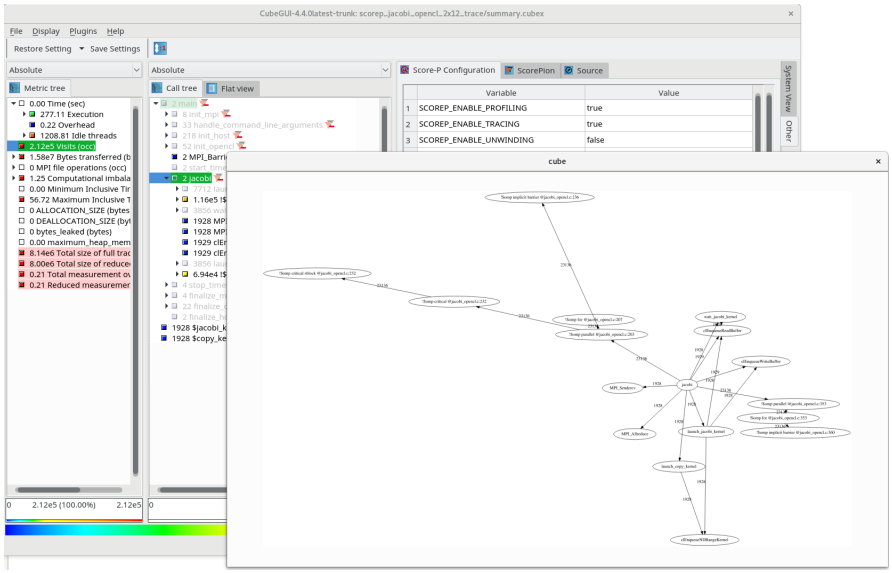
Jenga Fett

The *Jenga Fett* plugins allows to display metric values as bar charts along the system locations. It offers two modes: First, a stacked bar chart to display a whole metric sub-tree in one bar per process/thread as shown in Figure 14a, presenting the whole Time metric sub-tree. In the second mode, *Jenga Fett* allows to present multiple metrics as independent bars next to each other. The performance analyst can so easily spot correlations between the metrics. Figure 14b shows an example putting time and L2 cache misses (`PAPI_L2_TCM`) next to each other. It is clearly visible that processes with many cache misses have a high run-time while processes with a short run-time have only a few cache misses[4]. Another good use-case for this type of analysis is the *Scaling* plugin we presented in Section 3.

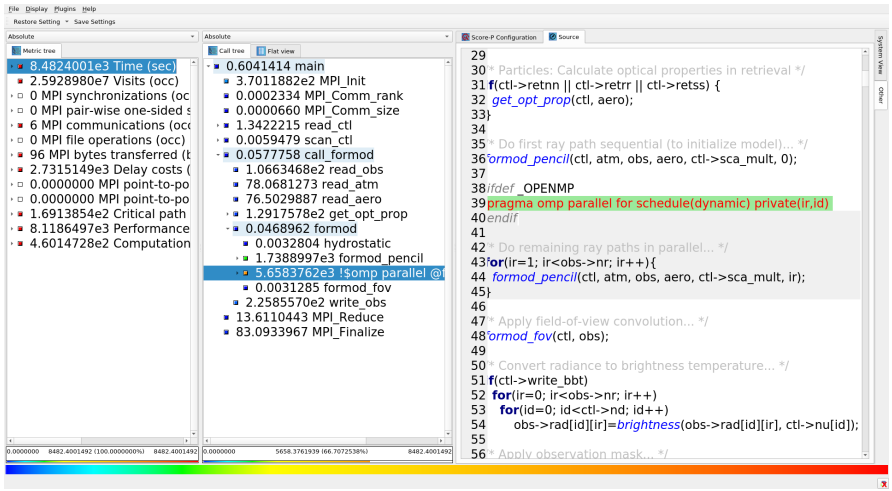
Advisor

Performance assessment of a parallel program can be a daunting task, as the causes of performance problems can be manifold. Major problems are a bad workload distribution, an inefficient communication scheme and a bad utilization of the system resources. In the course of the Performance Optimisation and Productivity Centre of Excellence (POP [1]) a methodology was developed to allow the performance analyst to acquire a standardized assessment of the code under investigation. This results in a hierarchal set of metrics [2] that quantify the relative impact of various performance factors. These metrics in general have a value between 0 and 1 (or 0% and 100% respectively), with a higher value being better.

[4] The application was a matrix-matrix-multiplication benchmark with alternating column-major and row-major outer loops

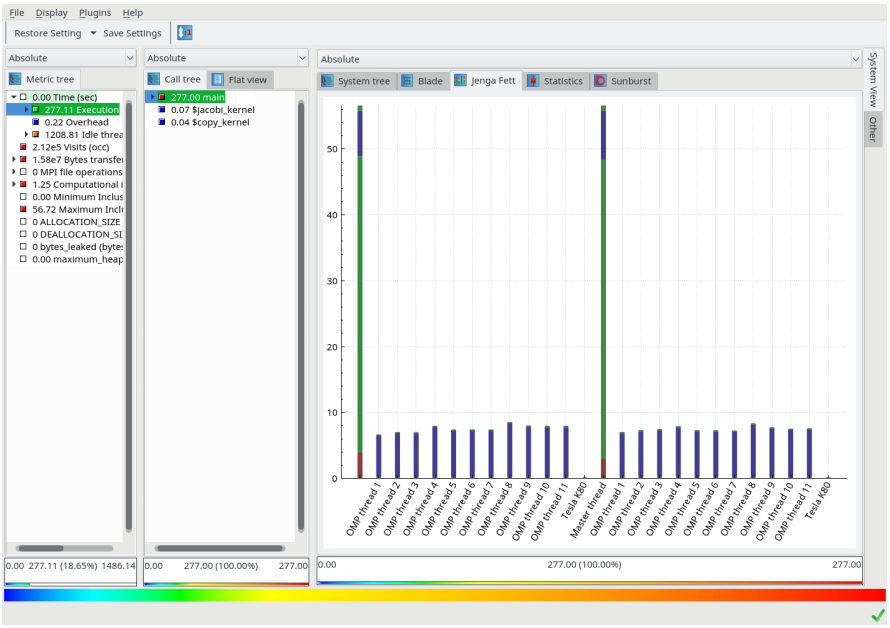


(a) Call Graph View

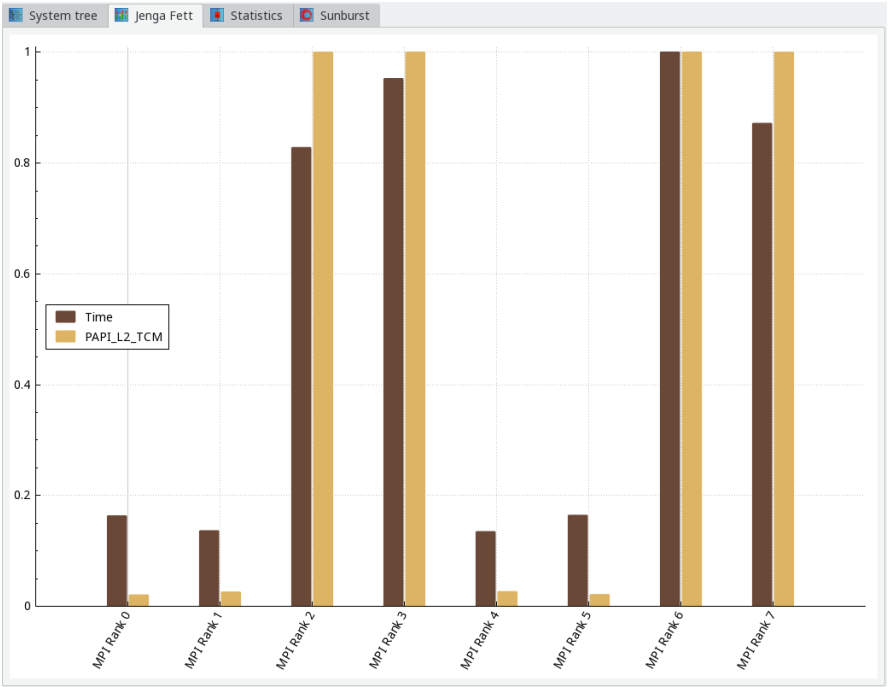


(b) Source Code View

Fig. 13: Screenshots of the *Call graph* plugin (a) and the *Source Code Viewer* plugin (b).



(a) *Jenga Fett* – Stacked bar plot



(b) *Jenga Fett* – Metric correlation plot

Fig. 14: Screenshots of the two modes of the *Jenga Fett* plugin. A stacked bar plot of Time (sub-)tree (a) and the correlation of multiple metrics (b). Here time and L2 cache misses are plotted next to each other and a clear correlation is visible.

The *Advisor* plugin makes the POP methodology metrics [2] available in the CubeGUI. Currently the following metrics are regarded:

- *Parallel Efficiency*: determines the performance loss when distributing computational work over the processes of the system. It is calculated as the product of *Load Balance* and *Communication Efficiency*.
- *Load Balance*: is the ratio of the average computation time (across all processes) and the maximum computation time (i.e. run-time without communication and synchronization).
- *Communication Efficiency*: is the maximum (across all processes) of the ratio between computation time and total run-time. *Communication Efficiency* identifies when code is inefficient because it spends a large amount of time communicating rather than performing useful computations. It is composed of two additional metrics that reflect two causes of excessive time within communication, *Serialisation Efficiency* and *Transfer Efficiency*
- *Serialisation Efficiency (SerE)*: measures inefficiency due to idle time within communications (i.e. time where no data is transferred).
- *Transfer Efficiency (TE)*: measures inefficiencies due to time in data transfer.

Further we report some hardware counter related metrics:

- Stalled resources: The ratio of cycles a processor was stalled and total CPU cycles.
- Instructions: The total number of ”useful” instructions being executed, i.e. not counting instructions in spin-wait phases.
- IPC: Instructions Per Cycle (IPC) is the number of useful instructions by CPU cycles and commonly used to determine the utilization of the processor. However, this metric alone can be misleading as the performance of an application strongly depends on the instructions being executed, i.e. a lower IPC can be better if vector instructions are used instead of scalar instructions.

The POP metrics can be calculated at any level of granularity - the whole application, a single kernel, or, with Cubes multiple selection feature, multiple kernels at the same time. Figure 15 shows an example of the *Advisor* plugin for the main computational routine of JURASSIC. Communication efficiency is very good as there is no MPI in this kernel and load balance is an issue as we have already seen in Figure 8b and Figure 12b. To present all metrics at once we need to merge at least two performance reports: A Scalasca trace analysis and a profile containing the PAPI counters `PAPI_TOT_INS`, `PAPI_TOT_CYC`, and `PAPI_RES_STL`. Without a trace analysis we have to omit the *Serialisation Efficiency* and *Transfer Efficiency* metrics.

6 Conclusion and future work

In this paper we gave an overview over the newly introduced CubeGUI plugin infrastructure. We described the broad spectrum of plugins and showed how they can

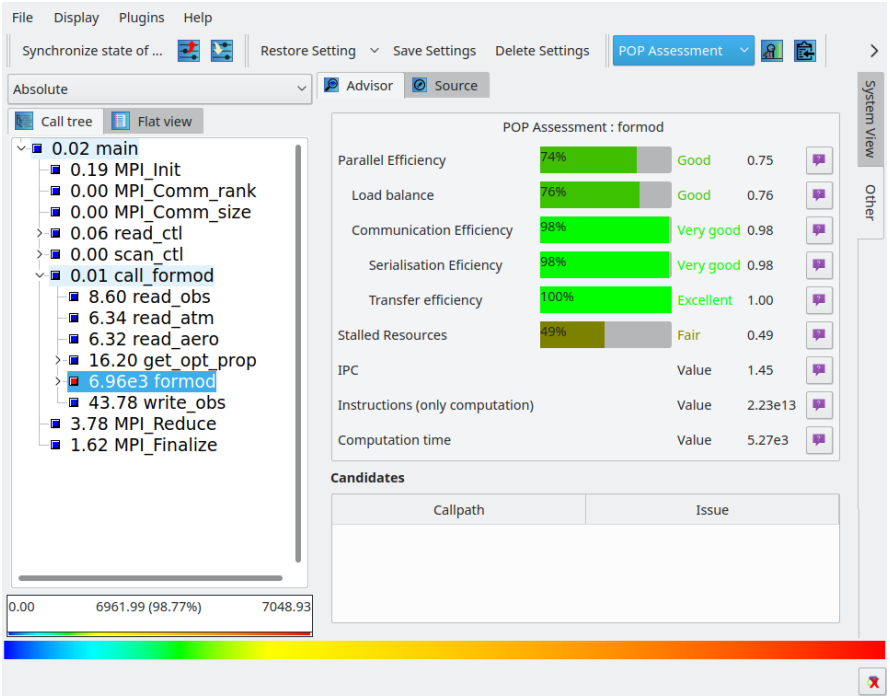


Fig. 15: Screenshot of the *Advisor* plugin showing the POP metrics for the main computational routine of JURASSIC.

help in everyday performance analysis. The enhancements of the system-tree help in the analysis of large-scale applications and several plugins increase the efficiency of the analysis by quickly pinpointing issues (like the *Advisor* plugin) or enabling novel types of analysis (like *Jenga Fett*). Context-free plugins make the often overlooked but incredibly useful Cube algebra utilities more accessible to non-expert analysts.

However, there is still a lot of ongoing and planned future work to do. First, we want to decouple more features from the core and provide them as plugins to keep the code as clean as possible. We plan to make plugins more powerful and versatile by enhancing the API and providing more mechanisms for plugins to communicate and interact with the CubeGUI. Performance is also an important topic, not only for applications but for performance analysis tools as well. We want to utilize the intra-node concurrency of modern CPUs to speed up the calculations within a plugin as well as the communication between plugins. As part of this, there is a current effort to make calculations asynchronous and distributed over smaller steps to incresase interactivity of the CubeGUI for larger experiments.

An ongoing development in Cube is the switch to a client-sever architecture, i.e. a server is running on the HPC system where the performance results are and a client

is running on the local machine of the performance analyst to utilize the hardware of HPC nodes and avoid transfering large amounts of data. The plugin infrastructure needs to be adapted to the architecture change in the GUI and we need a definition of modular plugins with a server-side" part and a "client-side" part of the plugin.

Of course we also strive to expand the list of available plugins – ideally including third-party developed plugins as well. We are looking at a tighter integration with other tools, both performance analysis tools (like for example Paraver [17]) and visualization tools (e.g. Paraview [6]). Performance analysis and tuning is still a very hard task and we want to ease that burden by providing an as comprehensive view as possible.

Acknowledgments

Parts of this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No 676553 and 824080.

References

1. Performance Optimisation and Productivity: A Centre of Excellence in HPC. https://pop-coe.eu/. Last access: 2019-09-16.
2. POP Standard Metrics for Parallel Performance Analysis. https://pop-coe.eu/node/69. Last access: 2019-09-16.
3. Scalasca website. https://www.scalasca.org.
4. Score-P website. https://www.score-p.org.
5. TeaLeaf Mini-app. https://uk-mac.github.io/TeaLeaf/.
6. James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *The visualization handbook*, 717, 2005.
7. Dirk Brömmel, Wolfgang Frings, Brian JN Wylie, Bernd Mohr, Paul Gibbon, and Thomas Lippert. The high-q club: Experience with extreme-scaling application codes. *Supercomputing Frontiers and Innovations*, 5(1):59–78, 2018.
8. 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 *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30 – September 2 2011*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2012.
9. Michael Frigge, David C. Hoaglin, and Boris Iglewicz. Some implementations of the boxplot. *The American Statistician*, 43(1):50–54, 1989.
10. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The SCALASCA performance toolset architecture. In *International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece*, pages 51–65, June 2008.
11. Jerry L. Hintze and Ray D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.
12. L Hoffmann and MJ Alexander. Retrieval of stratospheric temperatures from atmospheric infrared sounder radiance measurements for gravity wave studies. *Journal of Geophysical Research: Atmospheres*, 114(D7), 2009.
13. Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis toolset. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.
14. Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of the 5th Int'l Workshop on Parallel Tools for High Performance Computing, September 2011, Dresden*, pages 79–91. Springer, September 2012.
15. Allen D. Malony, Srinivasan Ramesh, Kevin Huck, Nicholas Chaimov, and Sameer Shende. A plugin architecture for the tau performance system. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 90:1–90:11, New York, NY, USA, 2019. ACM.
16. Dirk BR OMMEL, Wolfgang Frings, and Brian JN Wylie. Extreme-scaling applications 24/7 on juqueen blue gene/q. 2015.
17. 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, 1995.
18. Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: From performance report explorer to performance analysis tool. In *Proceedings of the International Conference on Computational Science, ICCS 2015, Computational Science at the Gates of Nature, Reykjavík, Iceland, 1-3 June, 2015*, pages 1343–1352, 2015.

19. 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.
20. John Stasko and Eugene Zhang. Focus+ context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings*, pages 57–65. IEEE, 2000.
21. Michael Stephan and Jutta Docter. Juqueen: Ibm blue gene/q® supercomputer system at the jülich supercomputing centre. *Journal of large-scale research facilities JLSRF*, 1:1, 2015.
22. Godehard Sutmann, Lidia Westphal, and Matthias Bolten. Particle based simulations of complex systems with mp2c: hydrodynamics and electrostatics. In *ICNAAM 2010: International Conference of Numerical Analysis and Applied Mathematics 2010*, volume 1281, pages 1768–1772. AIP Publishing, 2010.