Procedia Computer Science

Volume 51, 2015, Pages 1343–1352

ICCS 2015 International Conference On Computational Science

Cube v4 : From performance report explorer to performance analysis tool

Pavel Saviankou[1], Michael Knobloch[1], Anke Visser[1], and Bernd Mohr[1]

Forschungszentrum Jülich GmbH, Jülich Supercomputing Centre,
52425 Jülich, Germany
{p.saviankou,m.knobloch,a.visser,b.mohr}@fz-juelich.de

Abstract

Cube v3 has been a powerful tool to examine reports of the parallel performance tool Scalasca, but was basically unable to perform analyses on its own. With *Cube v4*, we addressed several shortcomings of *Cube v3*. We generalized the Cube data model, extended the list of supported data types, and allow operations with nontrivial algebras, e.g. for performance models or statistical data. Additionally, we introduced two major new features that greatly enhance the performance analysis features of Cube: Derived metrics and GUI plugins. Derived metrics can be used to create and manipulate metrics directly within the GUI, using a powerful domain-specific language called *CubePL*. Cube GUI plugins allow the development of novel performance analysis techniques and visualizations based on Cube data without changing the source code of the Cube GUI.

Keywords: Performance analysis, Call-tree profile, Derived metrics, DSL, GUI plugins

1 Motivation and Introduction

Performance analysis tools generate enormous amounts of data, especially for large-scale applications running on several thousands of compute cores. One of the major challenges for the tools is to identify and present the hot spots, i.e. the performance-critical parts of the program execution. So performance analysis tools have to be versatile and feature-rich, but on the same time easily accessible to the user.

For several years, the Cube framework [4] provided the data format and graphical user interface (GUI) for the performance analysis reports generated by Scalasca [5]. With the development of Score-P[7] as a community measurement system and profiler for multiple tools, a new version of Cube, *Cube v4*, was developed. This includes, among various tools and GUI improvements, a new file format. *Cube v3* used a single XML file, which was replaced by a tar archive consisting of an XML meta data file and several binary data and index files.

The set of tools distributed with the *Cube v3* framework only covered basic operations on Cube profiles and provided only limited data export and analysis capability. Due to its monolithic structure, Cube was difficult to maintain and extend, especially by third-party developers who wanted to perform novel types of analysis on Cube data.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.320

To ease performance analysis with Cube, we added various improvements in *Cube v4*. We extended the range of supported metric types, allow non-trivial data types for metrics and added non-trivial algebra, which goes beyond simple addition and subtraction. Data types do not have to be a single numeric value any more, but instead can be of a complex type, e.g. tuples of values, and might even be not numeric at all, e.g. strings. We introduced *derived metrics* in order to allow users to perform more complex performance analysis directly within Cube. To enable these, a powerful domain-specific language, called *CubePL*, was developed and integrated into Cube.

The *Cube v3* GUI offered little flexibility for presenting performance data other than showing trees with aggregated values. In order to provide more flexibility in data presentation we changed Cube's GUI architecture from a monolithic model to a plugin based one. Multiple plugins are provided by us and third parties. Users are also able to develop their own plugins to implement novel analysis techniques and visualizations. We are confident that all these changes in the Cube framework remove a significant number of drawbacks of *Cube v3* and strengthen Cube as a powerful tool for performance data analysis.

The rest of this paper is organized as follows: In section 2 we outline the Cube framework and highlight some changes in the transition from *Cube v3* to *Cube v4*. The next two sections are devoted to major enhancements in *Cube v4*. Cube's derived metrics and *CubePL* are presented in detail in section 3. Section 4 covers the Cube GUI plugin architecture and shows first examples of plugins. Related work for derived metrics in tools is discussed in section 5. Finally, we conclude the paper and give an outlook on future work in section 6.

2 Cube v4

Cube has been designed around a high-level data model of program behaviour called the *Cube performance space*. The Cube performance space consists of three dimensions: a metric dimension, a program dimension, and a system dimension. Each dimension of the performance space is organized in a hierarchy, as displayed in Figure 1. The usual operations while exploring Cube analysis reports are expanding and collapsing sub-trees to get the exclusive or inclusive values for the selected metric. Furthermore, the values shown in a pane are the aggregated values over the panes to its right.

The metric dimension contains a set of metrics, such as communication time or cache misses. This dimension is organized in an inclusive hierarchy, where a metric at a lower level is a subset of its parent. For example, communication time is a subset of execution time. The program dimension contains the program's call tree, which includes all the call paths onto which metric values can be mapped. The system dimension is organized in a multi-level hierarchy consisting of multiple levels. In *Cube v3* this was the fixed set machine, node, process, and thread. Each point (m, c, s) of the performance space can be mapped onto a number representing the actual measurement for metric m while the control flow of process/thread s was executing call path c. This mapping is called the *severity of the performance space*.

This general data model is common for *Cube v3* and *Cube v4*. However, we introduced several improvements and new features to it in *Cube v4*, not all of them directly visible by the user.

In contrast to *Cube v3*, the data relationship in the call tree is not any more always exclusive, but can vary from metric to metric, depending on its data density and algebraic properties. Now, it is possible to store the metric data in an inclusive format, i.e. a value for a call path includes the contribution of its sub-trees, or in an exclusive format, where every call path value corresponds to the call path itself, without its sub-trees.

The system dimension in *Cube v4* is organized in a more general manner - we do not restrict it to the four levels mentioned above - but allow an arbitrary depth in the system description. Instead of the fixed `machine` and `node` elements, *Cube v4* defines a generic `system tree node`, which can be anything from the whole machine down to a socket. Every system tree node can define `location`

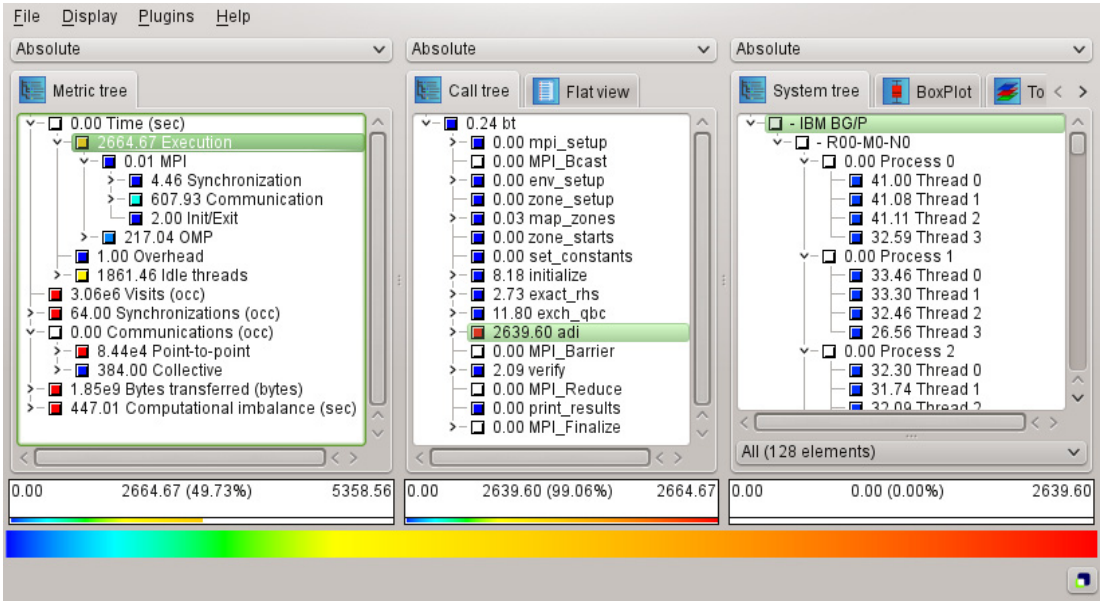


Figure 1: Cube screenshot as example of the Cube data model. The left pane shows the metric dimension, the middle pane the call tree and the right pane the system dimension.

`groups`, which usually represents processes[1]. Each location group defines `locations`, which are usually identical with execution threads. Every value within the Cube data model corresponds to one specific location.

Cube v4 extends the range of supported data types, which goes beyond plain numeric data types (integers, doubles, etc.), and now includes complex data types like scaling functions, histograms, or tuples describing statistical data. Moreover, *Cube v4* supports non-trivial algebra like `min` or `max` operators for numeric data types. As mentioned above, *Cube v4* allows to define data along the call tree dimension in an exclusive or inclusive format. This directly influences the formula for the calculation of the complementary value[2]. In the case of an exclusive metric, the corresponding calculation formula uses only the aggregation operator ”+”. However, in the case of an inclusive metric, also the inverse operator ”-” is used. Therefore, the data type plus the operator ”+” would build a `monoid` in first case and a `group` in the second one, where operator ”-” indicates an inverse value. When using a generalized operator ”+”, which does not have an inverse operator, the metric data has to be stored in an exclusive format. For example, the operator `max` doesn’t have an inverse form, so it can be only used with exclusive metrics. This aspect is especially important when redefining an operator ”+” in the context of derived metrics. Another remarkable extension of *Cube v4* is the possibility to define a `ghost metric`, which does not appear in the GUI or tools. Such a metric can be used as a container for raw measured data which is referred to in a derived metric, but has no meaning on its own.

[1] But can also be executions on a GPU or something similar.

[2] Inclusive value out of exclusive data, or exclusive value out of inclusive data

3 Cube Derived Metrics

One of the most remarkable features introduced with *Cube v4* is the possibility to perform a data transformation using so called `derived metrics`. With *Cube v3*, there were basically two ways to transform the data stored in a Cube performance report:

1. Writing an application that reads the Cube file, retrieves the data out of it, calculates desired metrics and exports result in one or another format. Visualization or further analysis of the results is then possible with additional tools like `R` or `gnuplot`. However, this usually required specialized tools which were not made available to the HPC community. Further, simultaneous analysis of the result of this tool and the measurement data is difficult because of the missing connection between the Cube GUI and the developed tool.

2. Another approach would be to extend the measurement system, record/calculate the desired metrics during the measurement, and store them next to the usual performance metrics, such as *execution time* or *function visits*, within the Cube report. In Scalasca, we implemented a flexible system for hardware counter metrics that way [11]. However, there are several drawbacks of this approach. First, it is necessary to reconfigure and to rerun the whole measurement for every new such "derived" metric (although of course multiple metrics could be recorded in one run). Further, not all metrics can actually be recorded during measurement or produce meaningful results when performing standard operations in the Cube GUI, such as calculation of inclusive or exclusive values of a call path by collapsing or expanding the corresponding sub tree.

The ideal approach would be the ability to extend an existing Cube profile by another metric, which delivers a meaningful result for every operation within the GUI, even if it goes beyond trivial aggregation. Therefore, with *Cube v4* we introduced so-called `derived metrics`. While Cube's predefined metrics are stored in the analysis report, the values of derived metrics are calculated on-the-fly when necessary according to user-defined arithmetic expressions formulated using a domain-specific language called *CubePL* (**Cube P**rocessing **L**anguage).

Derived metrics are also used by the Cube remapper, a tool for creation and proper nesting of a metric hierarchy in Cube performance reports. Initially, the profiler and trace analyzer create a raw list of metrics which is post-mortem processed by the Cube remapper. The user can configure the remapping by using a specification file containing the desired metric hierarchy.

3.1 Aggregation vs. Calculation

The main challenge in the definition of a derived metric in this manner is to ensure that it produces a meaningful result which doesn't collide with the normal interpretation of the values while exploring the measurement result in the Cube GUI. To demonstrate this aspect, let us assume that an application with the following call tree was executed:

```
main
 - foo
 - bar
```

where *main*, *foo* and *bar* are function calls, with *foo* and *bar* being called from *main*. Further, assume that the number of floating-point operations (`FLOP`) has been measured for every call path, as well as the respective execution time. In this example, exclusive values are stored for every call path.

To calculate the floating-point operations per second (`FLOPS`) as a derived metric for every call path, the naïve approach would be to define a new metric `FLOPS` and calculate its values for every call path using the formula $\frac{FLOP_c}{time_c}$, where c is either `main`, `foo`, or `bar`, and store the resulting values as a

data metric within the corresponding Cube profile. With this approach, however, a problem arises when the user would like to get an inclusive value of the metric `FLOPS` for the `main` call path. In this case, the Cube GUI would sum up all values of the metric `FLOPS` for every region and therefore deliver the result of the expression

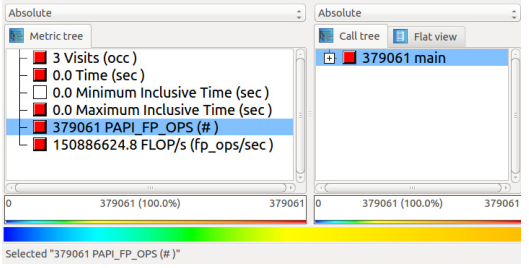
$$\frac{FLOP_{main}}{time_{main}} + \frac{FLOP_{foo}}{time_{foo}} + \frac{FLOP_{bar}}{time_{bar}} \tag{1}$$

However, this expression is the *sum of FLOPS of every region* instead of the intended *FLOPS of* `main`. So, the correct calculation would be

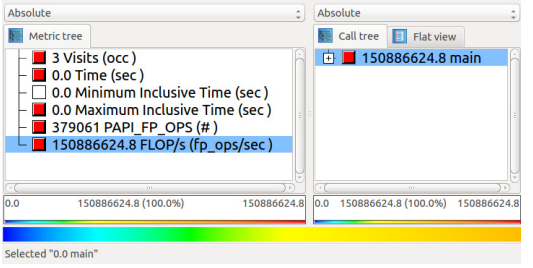
$$\frac{FLOP_{main} + FLOP_{foo} + FLOP_{bar}}{time_{main} + time_{foo} + time_{bar}} \tag{2}$$

which yields the desired floating-point operations per second for `main`. With the introduction of a derived metric in *Cube v4* it is possible to formulate metrics with the same or similar behaviour and perform the analysis within the Cube GUI, thus preventing the drawbacks mentioned above.

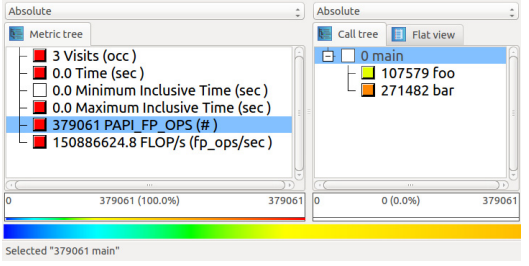
Figure 2 shows Cube screenshots for the `FLOPS` metric defined by equation (2). Inclusive values for FLOP and FLOPS are shown in Figures 2a and 2b, respectively. Figures 2c and 2d show the corresponding exclusive values, i.e. the values for `foo` and `bar`. It clearly shows the difference of equations (1) and (2). While the FLOP add up correctly, the FLOPS would not if equation 1 would be used.



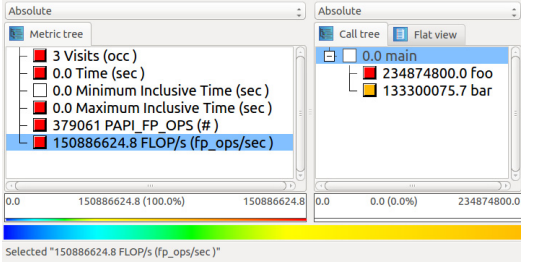
(a) FLOP inclusive



(b) FLOPS inclusive



(c) FLOP exclusive



(d) FLOPS exclusive

Figure 2: Cube screenshots showing the FLOPS metric

As shown in the previous example, the calculation of the $\frac{FLOP}{time}$ ratio should be performed as a last step – after the aggregation of the corresponding metric values for `FLOP` and `time` is done. We call this a *postderived* metric. In contrast, the case where the derived metric should be calculated before the aggregation of the involved metrics is called a *prederived* metric. This leads to three kinds of derived metrics within Cube:

Prederived metric - Its *CubePL* expression defines a value of a point in the Cube data model. This metric behaves in the same manner as any native data metric. There are two kinds of such metrics,

inclusive prederived metrics and **exclusive prederived metrics**.

Postderived metric - calculation of a *CubePL* expression is performed after all aggregations of referenced metrics are done. The example above should be formulated as a metric of this type.

3.2 Calculation Context

One of the most important things to consider when defining a derived metric in Cube is the *calculation context* for this metric. As mentioned in section 2, Cubes GUI displays [3] the data in the metrics panel as an aggregated value over the whole call tree and over the whole system tree. Thus, every value shown in the metric pane does not depend on a specific call path or system location, but only on one parameter, the metric being displayed. Analogous, the values in central pane are aggregated values over the whole system tree for a selected metric. So those values depend on two parameters, the selected metric and the call path being displayed. Finally, the values in most right panel are *discrete*. It means, for every value all three coordinates are defined: the selected metric, the selected call path, and the location in the system tree. Further, there is the question whether an inclusive or an exclusive value is being calculated. All these points define the context of the calculation.

3.3 CubePL

CubePL is a powerful domain-specific language to operate on Cube data. The basic syntax of *CubePL* is similar to many modern programming languages. It offers function calls, local and global variables and constants as well as the usual control flow statements. But its true power comes from the domain-specific functionality. *CubePL* defines special variables to interact with Cube data objects. First, there are `reserved` variables, which *CubePL* initializes during the creation of a Cube object. Examples are `${cube::#metrics}` or `${cube::filename}`. These variables have a global scope. Further, there are `automatic` variables, which are defined during every step of the calculation and contain information about the current calculation context. For example the variable `${calculation::callpath::id}` holds a numerical `ID` of the current call path. These variables have a local scope. This allows *CubePL* to use the value of other metrics of the Cube report in a *CubePL* expression. For this propose one uses a construction `metric::name(A,B)` where `name` is the unique name of the metric, and `A` and `B` are calculation context modifiers. `A` is a context modifier for the call tree and `B` is a context modifier for the system tree. There are three possible values for the context modifiers: `i` – enforces an inclusive evaluation, `e` – enforces exclusive evaluation and `*` – evaluation depends on state of call tree, i.e inclusive if collapsed and exclusive if expanded. These context modifiers can be omitted, in this case they default to `*`. For a detailed list of all possible calls and statements we are referring to [10].

To enhance the computational power of *CubePL* expressions, we introduced `parameterless defined-in-place`, or `lambda` function, calls. Listing 1 shows the basic syntax of such functions:

```
{
    [statement1];
    [statement2];
    ....
    [statementN];
    return [expression];
}
```

Listing 1: CubePL syntax for lambda functions

[3] In case Cube GUI shows the default configuration: the left pane displays the metric tree, central pane displays the call tree and the most right pane displays the system tree, see Figure 1

where `expression` yields the value of this function call. Usually, this is a value of a variable, which has been calculated in the sequence of statements 1 to N. Statements in the lambda function can be control flow constructs, loops, and memory assignments. An example of a lambda function is shown in Listing 2.

3.4 Examples

Here we give some examples of derived metrics, which might be useful for the performance analysis:

- The FLOPS metric presented above is a postderived metric with the *CubePL* expression

  ```
  metric::flop()/metric::time()
  ```

  Here we use the default evaluation which yields the results shown in Figure 2. Several hardware counter related metrics, like IPC (cycles per instruction) or similar, can be defined in the same manner.

- Another useful metric which calculates the average execution time per visit of a function is a postderived metric with the *CubePL* expression

  ```
  metric::time(i)/metric::visits(e)
  ```

  Here we use inclusive evaluation for the metric `time` and exclusive evaluation for metric `visits`. This means, it always shows the time including all functions called within the selected call path.

It is possible to use derived metrics to classify call paths and formulate metrics which represent only a part of another metric. Derived metrics of this kind are, for example, used in the Cube remapper. Listing 2 shows an example of an exclusive prederived metric, which gathers the time spent in the regions beginning on "`!$omp ordered @`"

```
{
${a}=0;
${regionid} = ${cube::callpath::calleeid}[${i}];
 if (${cube::region::name}[${regionid}]=~/^!\$omp ordered\s@/)
  { ${a}=metric::time(); }
 else
  { ${a}=0; };
 return ${a};
}
```

Listing 2: CubePL expression to determine the time spend in !$omp ordered

4 Cube GUI Plugins

Cube v4 (starting with version 4.3) provides a plugin interface for the Cube GUI. This interface allows the development of external tools for data representation and analysis, which are integrated into the Cube GUI. As no modification of the Cube source code is required, it's easy to develop and distribute new tools as plugins, independent of Cube's release schedule. Cube provides two different kinds of plugins:

- plugins that derive from the `CubePlugin` class depend on a loaded Cube file in the GUI. They can react on user actions, e.g. tree item selection, and may insert a context menu or add a new tab next to the tree views.

- plugins that derive from the `ContextFreeplugin` class are only active if no Cube file is loaded. These plugins create or modify Cube objects which can be loaded and displayed. That way the Cube command line tools – like `cube_merge` or `cube_diff` – can be integrated in the GUI.

Listing 3 shows a minimal example of a plugin definition. The function `treeItemIsSelected` connects this plugin with an user's action within the Cube Core GUI and allows the plugin to react on it. For further details we are referring to the Cube Plugin Development Guide [9]

```
class SimpleExample : public QObject, CubePlugin
{
 Q_OBJECT
 Q_INTERFACES( CubePlugin )
#if QT_VERSION >= 0x050000
 Q_PLUGIN_METADATA( IID "ExamplePlugin" ) // unique plugin name
#endif
public:
 // CubePlugin implementation
 virtual bool cubeOpened( pluginServices* service );
 virtual void cubeClosed();
 virtual QString name() const;
 virtual void version( int& major, int& minor, int& bugfix ) const;
 virtual QString getHelpText() const;

private slots:
 void treeItemIsSelected( TreeType  type, TreeItem* item );

private:
 pluginServices* service;
};
```

Listing 3: Basic Cube Plugin Structure

Several parts of the Cube GUI itself have been reimplemented as plugins, for example the topology and statistics views. Besides that, a wide range of different Cube plugins have already been developed that extend its functionality, or offers completely new analysis features. Some examples are:

- **Barplot and Heatmap:** These plugins work on special callpaths representing loop iterations. They show the dynamic behaviour of a loop as a barplot, applying one of the defined operations (min, max, avg), or present the data for locations and iterations as a colormap.

- **Callgraphplugin:** Uses Graphviz to generate a dependency graph for a selected metric.

- **Vampir [6] and Paraver [8] connector:** integrates *Cube v4* with these applications, for example to show the most severe instance of a performance problem in the timeline displays of these tools.

- **ScalingBehaviourExploration, PerformanceModeling, and HotspotHighlighting:** These plugins work on Cube files containing performance models, generated by "Extra-P" [3]. The ScalingBehaviourExploration plugin, shown in Figure 3, allows the detailed exploration and study of the scaling behaviour of a callpath, the HotspotHighlighting plugin evaluates the severity of the scaling behaviour of a callpath and highlights it in the Cube GUI. The PerformanceModelling plugin allows the interactive remodelling of the scaling models within Cube.

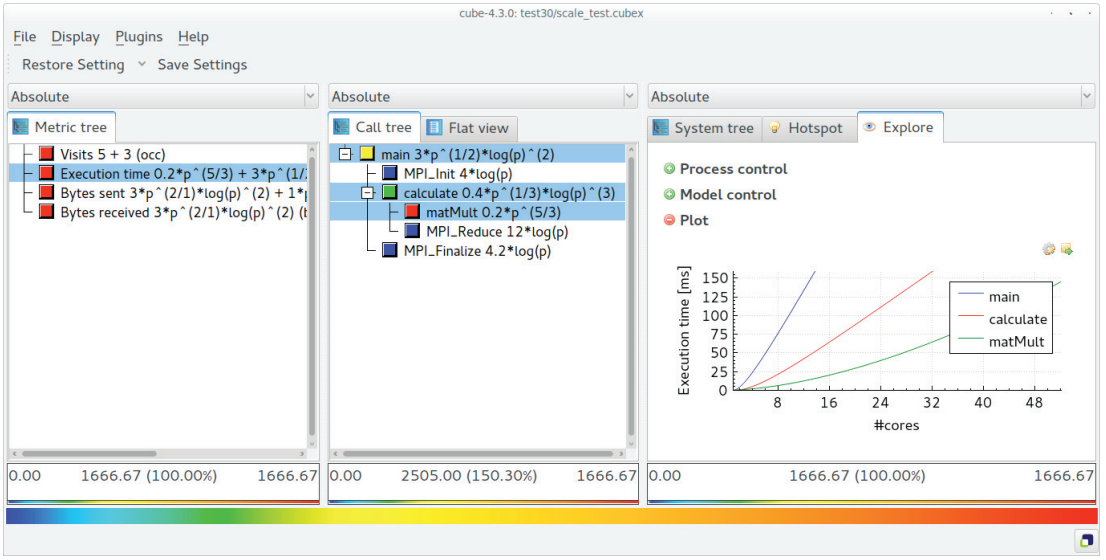


Figure 3: Cube screenshot showing the ScalingBehaviourExploration plugin. It also shows Cubes handling of performance models, i.e. scaling functions, as data types.

5 Related Work

Other performance analysis tools like ParaProf [2], Paraver [8], and Vampir [6] provide derived metrics for some time already. They typically allow to define simple arithmetic expressions build out of constants and already defined metrics. Paraver also provides powerful filtering and masking functions. The capabilities and features of *CubePL* go beyond this simple form of derived metrics. Due to the hierarchical analysis and visualization of performance data inside Cube, much more complex operations – which exceed simple arithmetic – and definitions are needed as explained in detail in Section 3.1. The usage of *CubePL* by the Cube remapper further requires possibilities to classify and filter metrics or metric data, which can also be benefitial for user-defined analyses.

6 Conclusion & Future Work

In this paper we presented the transition of the Cube framework from version 3 to version 4, which not only brought a new file format, but also several improvements in the GUI. We presented *Cube derived metrics* and *GUI plugins* and have shown how these features enhance the analysis capabilities of Cube. We will emphasize these new features in upcoming instances of our various bring-your-own-code tuning workshops, for example in the VI-HPS context [1], to increase awareness among our users and get feedback. We are looking into ways to provide a common repository for Cube plugins to collect and distribute both our own and third-party developed plugins.

Despite being an actively used software framework, the Cube framework is under constant development to implement new features requested by users, and to fix bugs found by our rigorous testing and by our user community. One of the main features that are currently under development is a client-server architecture for Cube. This will have many benefits: it will increase the scalability of the Cube GUI, it reduces the need to copy large Cube files to the users' own system and it will allow the creation of new front-ends, for example for mobile systems.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the Mont-Blanc 2 Project (www.montblanc-project.eu), grant agreement 610402 and the German DFG Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) under the Catwalk project. The authors would like to use this opportunity to thank the Scalasca and Score-P development teams for many fruitful discussions and remarks.

References

[1] VI-HPS Tuning Workshops web page. `http://www.vi-hps.org/training/tws/`.

[2] Robert Bell, Allen D Malony, and Sameer Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *Euro-Par 2003 Parallel Processing*, pages 17–26. Springer, 2003.

[3] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*, pages 1–12. ACM, November 2013.

[4] Markus Geimer, Björn Kuhlmann, Farzona Pulatova, Felix Wolf, and Brian J. N. Wylie. Scalable collation and presentation of call-path profile data with CUBE. In *Proc. of the Conference on Parallel Computing (ParCo), Aachen/Jülich, Germany*, pages 645–652, September 2007. *Minisymposium Scalability and Usability of HPC Programming Tools*.

[5] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The *Scalasca* performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[6] 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 (Proc. of the 2nd Parallel Tools Workshop, July 2008, Stuttgart, Germany)*, pages 139–155. Springer, July 2008.

[7] 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. 5th Parallel Tools Workshop (Dresden, Germany)*, pages 79–91. Springer, September 2012.

[8] 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.

[9] The Scalasca Development Team. CUBE 4.3.0 – Cube GUI Plugin Developer Guide. Distributed with the Cube framework.

[10] The Scalasca Development Team. Cube Derived Metrics. Distributed with the Cube framework.

[11] Brian J. N. Wylie, Bernd Mohr, and Felix Wolf. Holistic hardware counter performance analysis of parallel programs. In *Proc. of the Conference on Parallel Computing (ParCo), Malaga, Spain*, pages 187–194, September 2005.