D5.9 Paraver CUBE integration
Version 1.0

Document Information

<table>
<thead>
<tr>
<th>Contract Number</th>
<th>288777</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Website</td>
<td><a href="http://www.montblanc-project.eu">www.montblanc-project.eu</a></td>
</tr>
<tr>
<td>Contractual Deadline</td>
<td>PM24</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>PU</td>
</tr>
<tr>
<td>Nature</td>
<td>O</td>
</tr>
<tr>
<td>Authors</td>
<td>Harald Servat (BSC), Anke Visser (JSC)</td>
</tr>
<tr>
<td>Contributors</td>
<td>Judit Giménez (BSC), Pavel Saviankou (JSC)</td>
</tr>
<tr>
<td>Reviewer</td>
<td>Bernd Mohr (JSC)</td>
</tr>
<tr>
<td>Keywords</td>
<td>Performance tools, tools interaction, visualization, performance counters, source-code blaming</td>
</tr>
</tbody>
</table>

Notices:
The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 610402.

© Mont-Blanc 2 Consortium Partners. All rights reserved.
## Change Log

<table>
<thead>
<tr>
<th>Version</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0.1</td>
<td>Preliminary version by Harald Servat</td>
</tr>
<tr>
<td>V0.2</td>
<td>Merged versions from Pavel Saviankou and Anke Visser</td>
</tr>
<tr>
<td>V0.3</td>
<td>Modifications suggested by Judit Giménez</td>
</tr>
<tr>
<td>V0.4</td>
<td>Additional input from Anke Visser regarding Cube 4</td>
</tr>
<tr>
<td>V0.5</td>
<td>Review by Bernd Mohr</td>
</tr>
<tr>
<td>V1.0</td>
<td>Minor fixes and bump version to 1.0</td>
</tr>
</tbody>
</table>
Table of Contents

Executive Summary..............................................................................................................................................4

1 Introduction...................................................................................................................................................5
   1.1 Scalasca and CUBE .................................................................................................................................5
   1.2 The BSC performance tool-suite ...........................................................................................................6
      1.2.1 Description of the folding mechanism.............................................................................................6
      1.2.2 Workflow and analysis methodology ...............................................................................................8

2 Integration between CUBE and Folding .....................................................................................................9
   2.1 Description of the plugin interface from CUBE .....................................................................................9
   2.2 Description of the plugins created within the folding ..........................................................................10
      2.2.1 Description of the time-line visualization plugin ...........................................................................11
      2.2.2 Description of the source-code visualization plugin .......................................................................13

3 Usage example..............................................................................................................................................16
   3.1 BigDFT analysis .....................................................................................................................................17

4 Conclusions and future work .....................................................................................................................23

Acronyms and Abbreviations............................................................................................................................24

References........................................................................................................................................................24
Executive Summary

This document describes the work done to integrate the results and the representation mechanism from the Folding process developed at BSC into the Cube4 visualization tool developed at JSC. The Cube4 tool has been extended to augment its display and analysis capabilities via a plugin mechanism so that third party tools provide not only performance data to Cube but also new ways to represent the performance information within the Cube GUI. BSC has taken advantage of this extension to provide new visualization metaphors of its Folding mechanism to be able to represent in Cube4 the application progression in terms of performance and source-code between delimited code regions. This document also presents an example of the usage of this integration by describing the analysis of the BigDFT application.
1 Introduction

Supercomputers offer a vast amount of computational power but applications typically only reach a fraction of it. Performance analysis tools are pieces of software aimed at identifying performance bottlenecks so that application developers can understand the limitations of their applications and ultimately fix them.

The Folding [Folding2011] mechanism was developed at BSC and it displays the instantaneous performance for delimited regions of code of a given application. The Folding generates multiple reports to depict the progression for each delimited region as well as generating a Paraver [Paraver] trace-file. Recently, the Folding reports have been extended to explore the performance results using piece-wise linear regressions and attribute the phases to the source code [Folding2014].

Cube [Cube] is the profile-data visualization component for the Scalasca [Scalasca, Scalasca2] performance tool developed at JSC which allows exploring the performance from the source code perspective. It displays the performance data in three tree-like panels giving insight about the performance metrics, the application source-code and the system location. The tool aims at integrating other representation metaphors and to collaborate with other performance tools. As a consequence of effort, Cube4 [Cube4] has introduced a plugin infrastructure allowing third-party users (i.e. tool developers) to add new representation schemes into this tool and connecting it to new performance measurement sources.

This document reports the work done in the Folding tool to take advantage of the plugin infrastructure of the Cube4 tool. This work creates synergies between the two tools: the Cube visualization tool gains news representation mechanisms while the Folding results can be visualized in an easy manner. The rest of this section gives additional information regarding the related tools. Section 2 discusses the implementation of two plugins for the Cube-Folding interaction. Finally, Section 3 puts these plugins into practice by replicating the analysis and optimization of the BigDFT [BigDFT] application that was done in a meeting between the BSC performance analysis team and the BigDFT application developers at BSC during November 2,014.

1.1 Scalasca and CUBE

Scalasca is an open-source toolset under the BSD license developed at JSC that can be used to analyze the performance behavior of parallel applications and to identify opportunities for optimization. Target applications include simulation codes from science and engineering based on the parallel programming interfaces MPI and/or OpenMP. Scalasca, which has been specially designed for use on large-scale machines such as IBM Blue Gene and Cray XT, integrates runtime summaries suitable to obtain a performance overview with in-depth studies of concurrent behavior via event tracing.
Cube is a generic toolset for displaying and analyzing a multi-dimensional performance space consisting of three dimensions: performance metric, call path, and system resource. Each dimension can be represented as a tree, where non-leaf nodes of the tree can be collapsed or expanded to achieve the desired level of granularity. Cube and its associated Cube4 data format is used by Scalasca to produce an analysis report, which can be interactively explored with the Cube GUI or processed by various Cube command-line utilities.

1.2 The BSC performance tool-suite

The performance tool-suite developed at BSC is available as open-source under the LGPL license agreement. The suite is mainly composed by Paraver [Paraver] and Extrae [Extrae] as well as a set of satellite tools, such as the Folding and the ClusteringSuite [Clustering] that process Paraver trace-files implementing data-analytics techniques for performance analysis.

Extrae is the instrumentation package that generates time-stamped trace-files for multiple parallel programming paradigms (including MPI, OpenMP, and Pthreads, among many others). Extrae also includes sampling capabilities in order to complement the data collection. With respect to the collected data, Extrae can gather performance counters and call-stack information through the PAPI and libunwind libraries [PAPI, libunwind].

Paraver is a very flexible data browser that allows a detailed and powerful exploration of trace data. Programmable through configuration files, Paraver can visualize performance data via time-line displays (showing metrics per process or thread over time) or histogram displays (showing statistical data), independently from the programming model used.

The ClusteringSuite [Clustering] searches in the trace-file for computing regions (regions between the exit of a parallel programming call and the entry of the next parallel programming call) with similar characteristics.

The Folding implements a mechanism that takes advantage of instrumented and coarse-grain sampled information in order to report the detailed progression of performance metrics within delimited code regions. These regions can be manually delimited using the Extrae API or automatically detected by the ClusteringSuite.

1.2.1 Description of the folding mechanism

Performance monitoring tools, such as Score-P [Score-P] and Extrae rely on two collection techniques to invoke their performance monitors: instrumentation and sampling. Instrumentation refers to the ability to inject performance monitors into concrete application locations whereas sampling invokes the installed monitors periodically. Each technique has its advantages. The measurements obtained through instrumentation can be easily associated with the application source code structure while sampling allows a simple way to control the volume of the measurements captured. In any case, the granularity of the measurements provides valuable insight that cannot be easily determined a priori. Should analysts study the performance of an application for the first time, they may consider using a performance tool and instrument every routine or use high-frequency sampling rates in order to provide the most
detailed results. More often than not, these approaches lead to large overheads that impact on the application performance and thus alter the measurements gathered and, therefore, mislead the analyst.

The Folding mechanism overcomes the overhead by taking advantage of the repetitiveness found in many applications, especially within the HPC environment. This mechanism smartly combines instrumented and inexpensive coarse-grain sampled information that dilates the application runtime less than 5% on optimized binaries [Folding2011]. The results of the mechanisms include rich reports that show the instantaneous performance evolution and source-code progression within delimited regions of code. Within the Folding, BSC has evaluated different fitting algorithms to calculate the performance counter rates. Among the fitting alternatives studied, piece-wise linear regressions allows to automatically detect phases with uniform performance behavior.

1.2.1.1 Example

For exemplification purposes, we use the Stream benchmark [Stream]. This benchmark consists of a main loop that executes four different kernels sequentially where each kernel accesses different vectors. The code for this benchmark is summarized below in Table 1.

**Table 1: Summarized source-code for the Stream benchmark.**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:</td>
<td>int main ()</td>
</tr>
<tr>
<td>02:</td>
<td>{</td>
</tr>
<tr>
<td>03:</td>
<td>for (i=0; i&lt; NREPS; i++)</td>
</tr>
<tr>
<td>04:</td>
<td>{</td>
</tr>
<tr>
<td>05:</td>
<td></td>
</tr>
<tr>
<td>06:</td>
<td>for (j=0; j&lt;N; j++)</td>
</tr>
<tr>
<td>07:</td>
<td>c[j] = a[j]; /* copy */</td>
</tr>
<tr>
<td>08:</td>
<td>for (j=0; j&lt;N; j++)</td>
</tr>
<tr>
<td>09:</td>
<td>b[j] = scalar<em>c[j]; /</em> scale */</td>
</tr>
<tr>
<td>10:</td>
<td>for (j=0; j&lt;N; j++)</td>
</tr>
<tr>
<td>11:</td>
<td>c[j] = a[j]+b[j]; /* add */</td>
</tr>
<tr>
<td>12:</td>
<td>for (j=0; j&lt;N; j++)</td>
</tr>
<tr>
<td>13:</td>
<td>a[j] = b[j]+scalar<em>c[j]; /</em> triad */</td>
</tr>
<tr>
<td>14:</td>
<td></td>
</tr>
<tr>
<td>15:</td>
<td>}</td>
</tr>
<tr>
<td>16:</td>
<td>}</td>
</tr>
</tbody>
</table>

We have instrumented the benchmark’s main loop to delimit the region to be folded. That is we inserted calls to Extrae in lines 05 and 14 to delimit the loop body. Additionally, Extrae sampled the application every 25 ms and captured several performance counters in relation to the architecture efficiency in addition to the call-stack information. The results of the folding process for this execution on an Intel Core i7-2760QM running at 2.4 GHz is shown in Figure 1. This Figure represents the progression of the performance counters within the loop body. The black line refers to the instruction rate (or MIPS) and it is referenced on the right Y-axis, while the remaining lines refer to the L1D cache misses, L2D cache misses, L3 cache misses and branch mispredicts per instruction and they referenced on the left Y-axis. In this execution, the piece-wise linear regression detected three phase breaks (for a total of four phases) at times
14 ms, 28ms and 46 ms (shown as vertical dotted lines) according to the MIPS rate. The first phase ran approximately at 3600 MIPS and five out of every hundred instructions misses in L1D. As described in [Folding2014], it is possible to correlate these observed phases to particular regions of code (either routines or loops) considering the fact that applications are written as a set of routines that contain sequences of loops which may invoke nested loops or additional routines.

![Figure 1: Instantaneous metrics for the Stream benchmark obtained through the folding mechanism.](image)

### 1.2.2 Workflow and analysis methodology

The workflow starts with the generation of a trace-file of the application using the Extrae instrumentation package. Extrae is configured to capture parallel runtime activity (such as MPI, OpenMP, OmpSs) as well as to periodically capture the application behavior through sampling mechanisms. At these monitoring points Extrae captures the value of the performance counters and the call-stack. After generating the trace-file, it is passed to the ClusteringSuite. This tool explores the trace-file for similar computing regions (i.e. the code comprised between parallel runtime exit and the subsequent entry) and then uses a DBSCAN algorithm [DBSCAN] to group these regions according to their performance characteristics. The metrics typically used in this procedure refer to the execution rate (cycles-per-instruction [CPI], or inversely, instructions-per-cycles [IPC]) and the algorithmic complexity (the number of instructions executed and the ability of the algorithm to distribute the work among processes), but these metrics can be changed at user request. The grouping information is stored into the trace-file and this information is used by the Folding mechanism as the marker to delimit repetitive regions. This way the analyst can explore the application behavior using this tool without the necessity to know the application source-code structure beforehand. The workflow, and part of the methodology typically applied, finalizes by focusing on the detailed results provided by the Folding for those computing regions that represent most of the execution time. This ensures that if there exists optimization opportunities the return will be higher. With respect to the exploration of the Folding reports, we typically look for performance phases with low MIPS rate.
and then identify the nature of the performance bottleneck using the remaining performance counters captured. Since there is information regarding to the corresponding source-code, we simply match the application code with the identified performance bottlenecks.

2 Integration between CUBE and Folding

This section describes the interaction between the Cube visualization and the Folding mechanism. The section starts describing the Cube plugin interface. Then the section continues detailing the Folding mechanism and the methodology applied to explore first-time seen applications. Finally, the section reports two plugins that have been developed.

2.1 Description of the plugin interface from CUBE

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. It provides information about the original cube data, the data displayed in the trees, the order of trees and the selections inside. Plugins can react on user actions, e.g. tree item selection and can add its own controls e.g. by inserting a context menu or adding a new tab next to the tree views.

Cube defines two different kinds of plugins. 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.

Plugins that derive from the `CubePlugin` class depend on a Cube file loaded into the GUI. Examples for this type of plugins are the System Topology Plugin or the Statistics Plugin, which are part of the Cube installation. The plugins that are described in this document are also derived from `CubePlugin`.

```cpp
class CubePlugin : public PluginInterface
{
    public:
        virtual bool cubeOpened( PluginServices* service ) = 0;
        virtual void cubeClosed();
        virtual void version(int& major, int& minor, int& bug) const = 0;
        virtual QString name() const = 0;
        virtual QString getHelpText() const = 0;
};
```

Each plugin is identified by its name and its version. Only one plugin with the given name, the one with the highest version number will be loaded. The method `getHelpText` provides a text which is used to show a help menu item. The function `cubeOpened`, which is called if a user loads a cube file, is the starting point of the plugin. It provides an instance of `PluginServices` which is an interface to the Cube GUI and Library. It defines the following signals on which the plugin can react.
signals:
    void treeItemIsSelected( TreeType type, TreeItem* item);
    void contextMenuIsShown( TreeType type, TreeItem* item);
    void genericUserAction( UserAction action);
    void globalValueChanged( const QString& name);
    void orderHasChanged( const QList<DisplayType>& order);

It also offers access to the original cube data and the displayed data. A plugin can add its own menu, toolbar and tab widget. Some of the most important function of PluginServices are listened below.

class PluginServices : public QObject {
public:
    cube::Cube* getCube() const;
    QString getCubeBaseName() const;
    const QList<TreeItem*>& getTreeItems( TreeType type ) const;
    void selectItem( TreeItem* item, bool addSelection);
    void addTab( DisplayType type, TabInterface* tab );
    void addToolBar( QToolBar* toolbar );
    Qmenu *enablePluginMenu();
...
}

Cube offers a powerful and comprehensive interface for plugin developers. To document the whole API would go beyond the scope of this document. For more information, see the Cube Developer Guide [CubePluginDev], which is a tutorial for developing Cube plugins.

### 2.2 Description of the plugins created within the folding

In this project we have developed two plugins for Cube with different aims. The first plugin allows Cube to browse the Folding performance results. This plugin depicts the region instantaneous metrics using the same time-line metaphor as shown in Figure 1. This plugin avoids to manually dig into the multiple plots generated by the tool and use Cube as a single browser mechanism. The functionality of the second plugin eases the correlation between the application source-code and its performance. In this direction, both source-code and performance are arranged in a way so that the analyst easily can understand which regions of code ran slower or faster and also can identify what may be causing performance losses.

These plugins respond to some user Cube activities forwarded by the plugin infrastructure. Most notably, whenever the user clicks on a performance counter from the metric tree it toggles its visualization in both plugins. Also, when the user clicks on a different region in the Call-tree panel, the plugins automatically detect the change and update the visualization to show the selected region, if available.

Besides the plugins, it has been necessary to modify the Folding mechanism so that it generates a Cube file that serves as an entry point to these plugins. The Cube file contains information regarding the regions of interest, the definition of the performance metrics captured
during the application monitoring (executed instructions, cache misses, branch mispredictions), and for the source-code attribution plugin it contains references to the application code.

### 2.2.1 Description of the time-line visualization plugin

Currently, the most important outcome of the Folding mechanism is stored in comma-separated value (CSV) files that are originally processed by gnuplot. These files contain the associated performance counters for the original folded samples, the interpolation, and the derivative of the interpolation (i.e. the counter rate) for each combination of the available performance counters and code regions.

This plugin displays the instantaneous performance metrics generated by the Folding mechanism for every region. The outcome of this plugin is a plot representing the metrics that represent hardware performance counters along time (on the X-axis) in a similar way as depicted by gnuplot tool in Figure 2. To achieve this end, the plugin employs the QCustomPlot library [qcustomplot] to render these time-lines. QCustomPlot is a Qt C++ widget for plotting and data visualization which make it a natural candidate for this plugin because Cube4 also uses the Qt interface. It has no further dependencies and is fully documented. This plotting library focuses on making good looking, publication quality 2D plots, graphs and charts, as well as offering high performance for real-time visualization applications.

The plugin implements several methods from the CubePlugin superclass. In brief, the cubeOpened method is invoked whenever a Cube file is opened and whenever this occurs, the plugin searches for the CSV files and loads their contents into memory and also prepares the QCustomPlot widget as well as some additional necessary structures (see Table 2). The treeItemIsSelected method is invoked when the user clicks in any of the visible trees in the Cube panels (see Table 3). The plugin behavior depends on which panel has been clicked. If the user clicks on the metric tree, then the tool toggles the metric appearance in the widget and refreshes the output. However, if the user clicks in the calltree panel, then the widget changes which region to show then redisplay the selected performance counters for the selected region.

Since the MIPS rate is a good indicator of the achieved performance and it is typically available in current processors, we have decided to have this metrics from the start of the analysis (however, the analyst can toggle its appearance). This metric is represented always in black on the right Y-axis. Since the value of the remaining performance counters are directly dependent on the application activity and the processor architecture, it would be necessary to represent this remaining data in several scales. To avoid this, the remaining performance counters are depicted as the ratio of their respective counter rates and the instruction rate, i.e. report how many events occurred per instruction, and they are shown in the left Y-axis. The colors of the remaining metrics are assigned according to their selection order.
Figure 2: Cube plugin depicting the instantaneous performance computed by the folding mechanism.

Figure 2: Cube plugin depicting the instantaneous performance computed by the folding mechanism. shows a screenshot of the Cube visualizer with this plugin enabled. The plugin shows the performance achieved by the main loop of the Stream benchmark. Note that the plugin is attached to the rightmost panel. When capturing the screenshot, it was displaying the MIPS rate and several levels of the cache memory hierarchy (L1D in red, L2D in orange and L3 in green) per instruction as a response to the user selection.

Table 2: Code excerpt that responds to an open Cube File event in the time-line plugin.

```cpp
bool FoldingTimeline::cubeOpened( PluginServices* service )
{
    // Load csv data from the Folding package
    QString foldingdatafile = service->getCubeBaseName() + ".slope.csv";
    QFile* fmetrics = new QFile (foldingdatafile);
    if (fmetrics->exists())
    {
        if (fmetrics->open(QIODevice::ReadOnly|QIODevice::Text))
        {
            // Read data into containers
        }
    }
    // Initialize qtCustomplot infrastructure
    QPen p;
    p.setWidthF (WIDTH_PEN);
    p.setColor (QColor (255, 0, 0));
    plotPens.push_back (p);

    // Add this plugin to the system tab
    service->addTab (SYSTEM, this);
    // Add a hook for user activity on the tree
    connect(service, SIGNAL( treeItemIsSelected( TreeType, TreeItem*) ), this, SLOT( treeItemIsSelected( TreeType, TreeItem*) ) );
}
```
Table 3: Summarized code to respond on selecting an element in the Cube GUI for Time-line visualization plugin.

```cpp
void FoldingTimeline::treeItemIsSelected( TreeType type, TreeItem* item )
{
    QString ctrClicked;
    switch (type)
    {
    case METRICTREE:
        // If clicked on a metric, determine which metric
        // and toggle its presence in activeCounters set
        break;
    case CALLTREE:
        // The user clicked on a call?
        // Capture which call
        activeRegion = item->getTopLevelItem()->getName();
        break;
    default:
        default:
            return;
        break;
    }
    // Draw every the performance in the activeRegion for every
    // selected counter
    for (it = activeCounters.constBegin();
        it != activeCounters.constEnd();
        ++it)
        { // draw }
}
```

2.2.2 Description of the source-code visualization plugin

This plugin shows the source-code with the performance metrics side-by-side in order to let the
application analyst easily infer the reasons why the application code does not reach a better
performance. The idea behind this blaming derives from the fact that the phases identified by
the piece-wise linear regressions applied to the MIPS rate can be used to map the performance
with the source-code. Intuitively, the mapping is achieved by assigning the instruction rate to
the loops of the application instead of specific lines of the code because it is more natural due
to the fact that loops are where most of the application execution time is spent. The mapping
employs simple C/C++ and Fortran90 code parsers that delimit the loops in a source code and
helps assigning source code references to their container loop because in an ideal scenario,
each loop or routine would execute at a uniform performance. In this direction, the Folding
relies on the phases discovered by the piece-wise linear regression mechanism on the MIPS
rate because the instruction rate is the most commonly used to determine whether a code runs
efficiently or not. The resulting phases are used to attribute the metrics of the corresponding
phase to those loops that are contained in each phase. More details about these associations
are described in [Folding2014]. We note that [Folding2014] introduced experimental
modifications into an earlier version of Cube (3.x) that provided this mapping, but these
changes could not be merged into the current version of Cube (4.x) because of the large
differences between the two Cube versions. The introduction of the plugin infrastructure into Cube 4.3 lets us convert this code and ensure its availability to other Cube users.

The implementation of this plugin is similar to the previous one (with a summary of its implementation in Table 4 and Table 5). The plugin implements the same two call-backs as in the previous plugins which means that the plugin responds to loading new data into Cube as well as to user interaction. In this particular case, the Folding produces information data correlating the application source-code and the metrics observed. To represent this information we use a table in which columns shows different metrics as well as the source code, and every row represents a source-code line. Contrary to the previous plugin, the `cubeOpened` method is fairly simple and only initializes the table widget. The second method keeps track of the user activity. If the user clicks on a performance metric then it toggles its visualization in the table. If the user clicks on the call-stack tree, then the plugin searches for the corresponding source-code (which is pointed in the Cube file) and also for its metrics, and if it is available it then displays its contents in the table. Finally, the table widget also responds to user activity through the `onTableClick` method. If the user clicks on a column that represents a performance metric, then the metrics and the source-code are colored according to a Cube gradient. This helps the user to easily identify high and low values within the table.

Figure 3 shows the outcome of this plugin in the rightmost panel when using the same data for the Stream benchmark as shown in Figure 2. The table contents is as follows: rows refer to source code lines while columns refer to phase, selected metrics, and line numbers (second column from the right) or the code itself (rightmost column). More precisely, Figure 3 shows eight columns. From left to right: Phase number, MIPS, #Occurrences (i.e. the number of observed samples per line), PAPI_L1_DCM/ins, PAPI_L2_DCM/ins, PAPI_L3_DCM/ins, Line and the application Code. Note that this table shares the same counters as in Figure 2 because this screenshot was generated after changing the Cube plugin but not clicking on any other metric. Additionally, the heads match the coloring of the time-line in order to ease the correlation between them. The table also responds to clicks on cells that refer to performance metrics (that is, all but the last two on the right). If a user clicks on a cell, then the plugin associates to each row a background color noting its severity within the range defined by the selected metric and following the Cube severity color gradient. In the Figure, the user clicked on the MIPS rate (notice that it is shown in bold face in the heading row). In this Figure, we note that the third kernel (add) achieves the worst instruction rate (3,206 MIPS, shown in blue) and approximately 6% of the instructions miss at L1D, while the second kernel (scale) achieves the best instruction rate (4,250 MIPS, shown in red) and approximately 4% of the instructions miss at L1D.
Table 4: Summarized code to respond on loading a Cube File for the source-code correlation plugin.

```cpp
bool FoldingSourceCode::cubeOpened( PluginServices* service )
{
    // Prepare the table widget
    sourceCodeWidget = new QWidget;
    QVBoxLayout *layout = new QVBoxLayout();
    sourceCodeTable = new QTableWidget;
    ...

    // Add this plugin to the system tab
    service-&gt;addTab (SYSTEM, this);

    // Do some setup
    colorBlack = QColor (0, 0, 0);
    colorWhite = QColor (255, 255, 255);
    colorPalette.push_back (QColor (255, 0, 0));
    ...
    fontSourceCode = QFont("Courier 10 Pitch", 10);

    // Answer to tree activity
    connect( service, SIGNAL( treeItemIsSelected( TreeType,
                        TreeItem* ) ), this, SLOT( treeItemIsSelected( TreeType,
                        TreeItem* ) ) );

    // Answer to table activity
    connect (sourceCodeTable, SIGNAL( cellClicked (int, int) ),
             this, SLOT(onTableClick(int,int)));
}
```

Figure 3: Cube plugin correlating the instantaneous performance computed by the folding mechanism and the associated source-code.
Table 5: Code snippet for the source-code associated plugin when the user selects a new tree in the Cube GUI.

```cpp
void FoldingSourceCode::treeItemIsSelected( TreeType type, TreeItem* item )
{
    // Load data for the active region into the table
    QString ctrClicked;
    switch (type)
    {
    case METRICTREE:
        // If clicked on a metric, determine which metric
        // and toggle its presence in activeCounters set
        break;
    case CALLTREE:
        // The user clicked on a call?
        // Capture which call
        activeRegion = item->getTopLevelItem()->getName();
        break;
    default:
        return;
        break;
    }
    // Display the table
}
```

```cpp
void FoldingSourceCode::onTableClick (int row, int column)
{
    // Update the min/max values for color rendering according to
    // the column clicked.
}
```

3 Usage example

This section serves as an analysis example to demonstrate the capabilities of the Folding and the integration of its results within the Cube visualizer. This application was analyzed during a meeting at BSC in November 2014 between the BSC performance tools team and some BigDFT application developers. This document reproduces that analysis by the means of the new Cube plugins detailed here.

Regarding the application, BigDFT is developed at INAC/CEA and it is one of the selected applications within the Mont-Blanc/Mont-Blanc2 project. The application is a DFT massively parallel electronic structure code based on density functional theories using a wavelet basis set. Wavelets form a real space basis set distributed on an adaptive mesh (two levels of resolution in the BigDFT implementation). The results described here extend the analysis previously described in [Folding2014].

BigDFT was executed in the JuQueen supercomputer using 1,024 MPI processes. JuQueen is a BlueGene/Q system running IBM PowerPC A2 processors at 1.6 GHz. The application was compiled using the flags `-O2 -qtune=qp -qarch=qp -qreport`. Notice that BigDFT does not allow compiling it with `-O3` because the results do not converge in this
system. With respect to the performance gathering, we used the default MPI instrumentation from Extrae 2.5.2 and sampled the application at 20 Hz rate (which is far below the 100Hz that gprof uses by default), thus ensuring a negligible overhead during the application run.

### 3.1 BigDFT analysis

![Figure 4: MPI activity in BigDFT as reported by Paraver.](image)

A preliminary analysis of the application considering its MPI activity on a subset of the processes is shown in Figure 4. This screen-shot shows a time-line of the MPI calls. The X-axis represents time, the Y-axis represents the processes, and the color on each process at a given time depends on the parallel call (and also the pale blue represents outside MPI). From this time-line, Paraver can also calculate a profile per MPI call resulting in different interesting metrics. First, the application spends less than 8% in average in MPI calls, which means that the application is executing user code for more than 92% (as shown on the left column of the plot). There are also low variations in the execution of the user code, which means that the execution was well-balanced.

As stated earlier, in order to avoid the user instrumenting every routine in the application code, we relied on a clustering tool that groups computation regions according to their performance characteristics. Figure 5 shows the results of this tool when applied to the obtained BigDFT trace-file. The Figure is a scatter-plot in which the Y-axis represents the number of instructions while the X-axis represents the CPI and they indicate the algorithmic complexity and the execution rate, respectively. According to this plot, regions that are on the right are slow and regions that are at the top involve lots of instructions to execute.
Additionally, the clustering tool also generates a Paraver trace-file with information regarding the progression of the identified clusters. Figure 7 shows a Paraver screen-shot depicting the progression of these regions along time. This time-line indicates that the application has a very strong repetitive pattern mainly composed by the green and yellow regions. It also states that the application is SPMD because the behavior is the same among processes. The Paraver tool indicates that Clusters 1 (green), 2 (yellow) and 3 (red) represent respectively approximately 35%, 32% and 5% of the execution time as shown in Figure 6.
The typical methodology consists on focusing on those computing regions that take most time to execute because the return would be higher if there’s any chance to improve them. Since Cluster 1 was already explored in [Folding2014], during the meeting we focused the performance of the next Cluster in terms of execution time (Cluster 2). This computation region typically executed 2.376 million instructions at 550 MIPS, so each invocation lasted approximately 4.305s. Figure 8 shows the instantaneous metrics (MIPS and L1D cache misses/instruction) for Cluster 2. The results indicate that the region progresses through five phases. From these phases we noticed that the third phase exhibits a very bad behavior because it ran below 200 MIPS and approximately one out every three instructions misses in L1D, which is likely to be the reason of such a low instruction rate.

Figure 8: The instantaneous metrics for Cluster 2 shown in the plugin for Cube.

Figure 9 shows the source-code correlation plugin while focusing on the code associated to the phases 2 and 3 which respectively achieve the highest (shown in red) and lowest (shown in blue) MIPS rates. For this analysis we focused on Phase 3 and it referred to the application routine \texttt{conv.kin.y.new} from the \texttt{convolute.optim slab.f90} file and the tool pointed to a code that was manually unrolled eight times (in lines 1,166-1,176). This unrolled loop was continuously accessing to memory with very long strides (last dimension on the 3D matrix \texttt{x}).
Interestingly, the peeling code of the unrolled loop (which is below, not shown in Figure) showed smaller strides because the loop iterates over the 2nd dimension of x.

The unrolling observed in the application was an ad-hoc solution according to the application developers, thus we later modified the code to remove this unrolling and observe whether the application performance changes. After these modifications, we re-ran the application using the same experiment and the scatter-plot of the clustering tool looks like Figure 10. We outline that the group of points moved to the left (meaning that the computation ran faster) and a bit to the top (executed more instructions due to the increased number of control instructions). More precisely, the group of computation regions executed 2.538 million instructions at 650 MIPS, so each invocation lasted approximately 3.893 s which is 9.5% less execution time than the original version.

Figure 9: The Cube plugin showing the associated code to two phases within Cluster 2. Phase 2 (top) reaches the highest MIPS rate and it is shown in red, while Phase 3 (bottom) achieves the lowest MIPS rate and it is shown in blue.
The instantaneous metrics for this code region is depicted in Figure 11. The time-line shows an increased instruction rate (approximately 450 MIPS) in Phase 3, which is a 125% improvement compared to the original version. This increased performance is likely to respond to the better cache hierarchy usage in the new version because less than 2% of the instructions miss at L1D. Finally, with respect of the application code (see Figure 12) we noticed that the source-code plugin points only to the non-unrolled loop in lines 1,202-1,210 from the routine conv_kin_y_new. After applying these modifications to the source code, the application took 3.5% time less to execute.

Figure 10: Clustered results for the computation regions of the BigDFT execution after the code modification.
Figure 11: The instantaneous metrics for Cluster 2 shown in the plugin for Cube after modifying the application code.

Figure 12: The Cube plugin showing the associated code to phase 3 within Cluster 2 after modifying the application code.
4 Conclusions and future work

This document has presented the extensions done by JSC into the Cube to allow interactions with third party tools. The extensions allow Cube to augment its display and analysis capabilities through a plugin mechanism. In this direction, and also reported in this document, BSC has created two plugins to display the results from the Folding tool. These plugins add two new visualization metaphors into Cube (time-line performance progression, and source-code and performance correlation). This document has also revisited the analysis done to the BigDFT application by the BSC performance analysis tools team to show the analysis workflow using the newly developed plugins.
Acronyms and Abbreviations

- DFT Density Functional Theories
- SPMD Single Program Multiple Data

References


[Stream]  https://www.cs.virginia.edu/stream


[libunwind] https://savannah.nongnu.org/projects/libunwind