CUBE - User Manual

Generic Display for Application Performance Data

Version 2.0 / June 1, 2005

Fengguang Song, Felix Wolf

Contents

Abstract

CUBE is a generic presentation component suitable for displaying a wide variety of performance metrics for parallel programs including MPI and OpenMP applications. Program performance is represented in a multi-dimensional space including various program and system resources. The tool allows the interactive exploration of this space in a scalable fashion and browsing the different kinds of performance behavior with ease. CUBE also includes a library to read and write performance data as well as operators to compare, integrate, and summarize data from different experiments. This user manual provides instructions of how to install CUBE, how to use the display, how to use the operators, and how to write CUBE files.

1 Introduction

CUBE (CUBE Uniform Behavioral Encoding) is a generic presentation component suitable for displaying a wide variety of performance metrics for parallel programs including MPI [2] and OpenMP [3] applications. CUBE allows interactive exploration of a multidimensional metric space in a scalable fashion. Scalability is achieved in two ways: hierarchical decomposition of individual dimensions and aggregation across different dimensions. All metrics are uniformly accommodated in the same display and thus provide the ability to easily compare the effects of different kinds of program behavior.

CUBE has been designed around a high-level data model of program behavior called the CUBE *performance space*. The CUBE performance space consists of three dimensions: a metric dimension, a program dimension, and a system dimension. The metric dimension contains a set of metrics, such as communication time or cache misses. 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 contains all the control flows of the program, which can be processes or threads depending on the parallel programming model. Each point (m,c,l) of the space can be mapped onto a number representing the actual measurement for metric m while the control flow l was executing call path c. This mapping is called the *severity* of the performance space.

Each dimension of the performance space is organized in a hierarchy. First, the metric dimension is organized in an inclusion hierarchy where a metric at a lower level is a subset of its parent, for example, communication time is below execution time. Second, the program dimension is organized in a call-tree hierarchy. Flat profiles can be represented as multiple trivial call trees consisting only of a single node. Finally, the system dimension is organized in a multi-level hierarchy consisting of the levels: machine, SMP node, process, and thread.

CUBE also includes a library to read and write instances of the previously described data model in the form of an XML file. The file representation is divided into a *metadata* part and a *data* part. The metadata part describes the structure of the three dimensions plus the definitions of various program and system resources. The data part contains the actual severity numbers to be mapped onto the different elements of the performance space.

The display component can load such a file and display the different dimensions of the performance space using three coupled tree browsers (Figure 1). The browsers are connected so that the user can view one dimension with respect to another dimension. For example, the user can click on a particular metric and see its distribution across the call tree. If the CUBE file contains topological information, the distribution of the performance metric across the topology can be examined using the CUBE topology view. Furthermore, the display is augmented with a source-code display that can show the exact position of a call site in the source code.

As performance tuning of parallel applications usually involves multiple experiments to compare the effects of certain optimization strategies, CUBE includes a new feature designed to simplify cross-experiment analysis. The CUBE algebra [4] is an extension of the framework for multi-execution performance tuning by Karavanic and Miller [1] and offers a set of operators that can be used to compare, integrate, and summarize multiple CUBE data sets. The algebra allows the combination of multiple CUBE data sets into a single one that can be displayed like the original ones.

The following sections explain how to install CUBE, how to use the display, how to create CUBE files, and how to use the algebra and other tools.

2 Installation

CUBE is available as a source-code distribution for UNIX platforms. You can download CUBE from:

`http://icl.cs.utk.edu/kojak/cube/`

Building CUBE requires the XML parser libxml2 and the GUI toolkit wxWidgets.

2.1 Installing CUBE

The full installation includes the CUBE library to create CUBE files, and the CUBE display component to display their contents.

1. `gunzip cube-2.0.tar.gz | tar xvf`
2. `cd cube-2.0`
3. Edit `Makefile.defs`
   - Set the variable `PREFIX` to your desired installation path.
   - Depending on the platform, select and uncomment a specific block corresponding to your operating system. Available options are LINUX, AIX, IRIX, and SOLARIS. To customize the compiler setting, please edit the following variables:

     `CCC`: C++ compiler. Note that the compiler must be the same as the compiler used for compiling WXWIDGETS, unless you build the library only.

     `CCFLAGS`: C++ compiler options

     `LDFLAGS`: Linker options

     `AR`: Archive tool (e.g., `ar` or `CC`)

     `ARFLAGS`: Archive tool options

4. `make`

5. `make install`

2.2 Installing CUBE Library only

The partial installation will build and install only the CUBE library on your system. This is intended for users who just need to create CUBE file, but need not display it on their machines.

1. Same as steps of 1 to 3 described in the above section.

2. `make lib`

3. `make install-lib`

2.3 License

This software is free but by downloading and using it you automatically agree to comply with the license agreement. You can read the file `LICENSE` in the distribution for precise wording.

2.4 Libraries Required

Both libraries listed below are necessary for using the CUBE display component. For those users who need the CUBE library only, only libxml2 is required to be installed.

- **libxml2** (2.5.6), which is an XML C parser and toolkit developed for the Gnome project. It is pre-installed on many systems. Please refer to the libxml2 web page for details:

  `http://xmlsoft.org/`

- **wxWidgets** (2.4.2), which is a cross-platform C++ framework for writing advanced GUI applications using native controls. Please refer to the wxWidgets web page for details:

  `http://www.wxwidgets.org/`

2.5 Support

If you have any question or comments you would like to share with the CUBE developers, please send e-mails to `kojak@cs.utk.edu`.

3 Using the Display

This section explains how to use the CUBE display component. After a brief description of the basic principles, different components of the GUI will be described in detail.

3.1 Basic Principles

The CUBE display consists of three tree browsers, each of them representing a dimension of the performance space (Figure 1). The left tree displays the metric dimension, the middle tree displays the program dimension, and the right tree displays the system dimension. The nodes in the metric tree represent metrics. The nodes in the program dimension can have different semantics depending on the particular view that has been selected. In Figure 1, they represent call paths forming a call tree. The nodes in the system dimension represent machines, nodes, processes, or threads from top to bottom.

Users can perform two types of actions: selecting a node or expanding/collapsing a node. At any time, there are two nodes selected, one in the metric tree and the other in the call tree. It is not possible to select a node in the system tree.

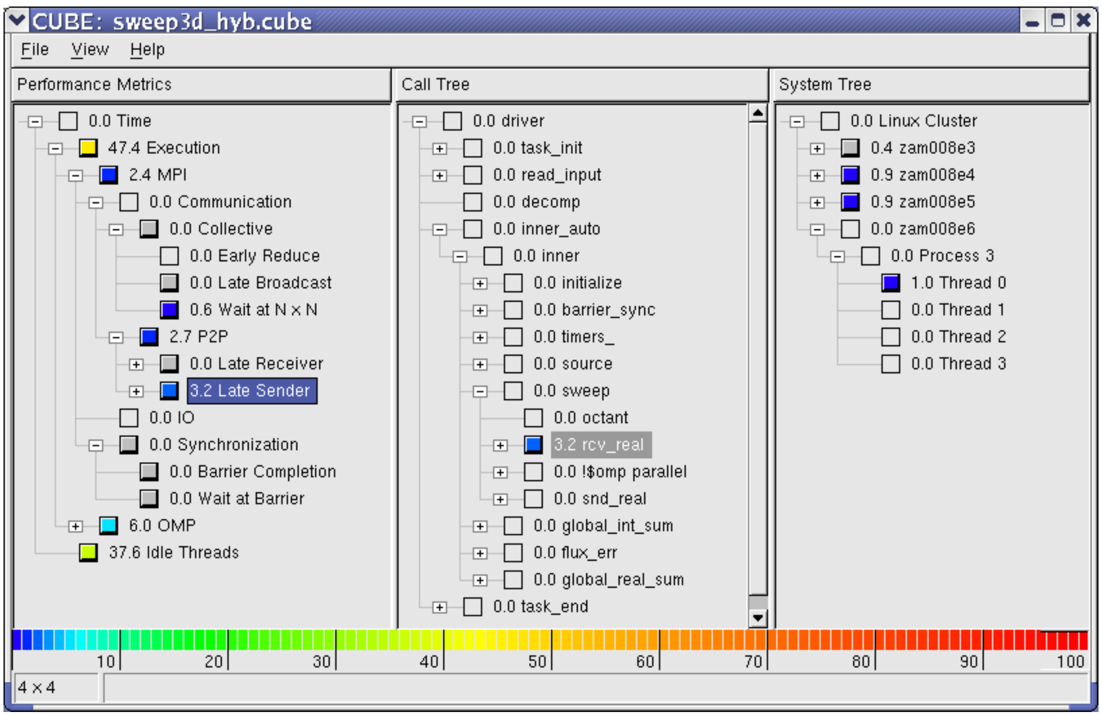

Figure 1: CUBE display window.

Each node is associated with a metric value, which is called the *severity* and is displayed simultaneously using a numerical value as well as a colored square. Colors enable the easy identification of nodes of interest even in a large tree, whereas the numerical values enable the precise comparison of individual values. The sign of a value is visually distinguished by the relief of the colored square. A raised relief indicates a positive sign, a sunken relief indicates a negative sign. Figure 6 shows nodes with positive and negative signs. Negative values can appear as a result of applying the algebra's difference operator (Section 4.1) to two data sets.

A value shown in the metric tree represents the sum of a particular metric for the entire program, that is, across all call paths and the entire system. A value shown in the call tree represents the sum of the selected metric across all processes or threads for a particular call path. A value shown in the system tree represents the selected metric for the selected call path and a particular system resource. Briefly, a tree is always an aggregation of all of its neighbor trees to the right. If there are multiple call trees, CUBE has two options to compute the overall severity for a particular metric. Either it can calculate the sum of all call trees (i.e., their root nodes in collapsed state) or their maximum. If CUBE is unable to determine the correct mode it will ask the user.

Note that all the hierarchies in CUBE are inclusion hierarchies, meaning that a child node represents a part of the parent node. For example, the metric hierarchy might display cache misses as a child node of cache accesses because the former event is a subset of the latter event. Similarly, in Figure 2 the call path *main* contains the call paths *main-foo* and *main-bar* as child nodes because their execution times are included in their parent's execution time.

The severity displayed in CUBE follows the principle of *single representation*, that is, within a tree each fraction of the severity is displayed only once. The purpose of this display strategy is to have a particular performance problem to appear only once in the tree and, thus, help identify it more quickly. Therefore, the severity displayed at a node depends on the node's state, whether it is expanded or collapsed. The severity of a collapsed node represents the whole subtree associated with that node, whereas the severity of an expanded node represents only the fraction that is not covered by its descendants because the severity of its descendants is now displayed separately. We call the former one *inclusive* severity, whereas we call the latter one *exclusive* severity.

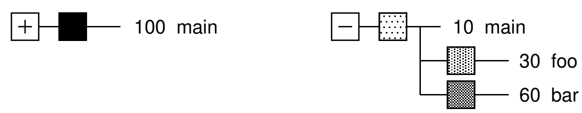

Figure 2: Node of the call tree in collapsed or expanded state.

For instance, a call tree may have a node *main* with two children *main-foo* and *main-bar* (Figure 2). In the collapsed state, this node is labeled with the time spent in the whole program. In the expanded state it displays only the fraction that is spent neither in *foo* nor in *bar*. Note that the label of a node does not change when it is expanded or collapsed, even if the severity of the node changes from exclusive to inclusive or vice versa.

3.2 GUI Components

The GUI consists of a menu bar, three tree browsers, a color legend, and a status bar. In addition, some tree browsers provides a context menu associated with each node that can be used to access node-specific information.

3.2.1 Tree Browsers

The tree browsers are controlled by the left and right mouse buttons. The left mouse button is used to select or expand/collapse a node. The right mouse button is used to pop up a context menu with node-specific information, such as online documentation. Context menus are only available for the metric and program trees.

A label in the metric tree shows a metric name. A label in the call tree shows the last callee of a particular call path. If you want to know the complete call path, you must read all labels from the root down to the particular node you are interested in. After switching to the module-profile or region-profile view (see below), labels in the middle tree denote modules or regions depending on their level in the tree. A label in the system tree shows the name of the system resource it represents, such as a node name or a machine name. Processes and threads are usually identified by a number, but it is possible to give them specific names when creating a CUBE file. The thread level of single-threaded applications is hidden. Note that all trees can have multiple root nodes.

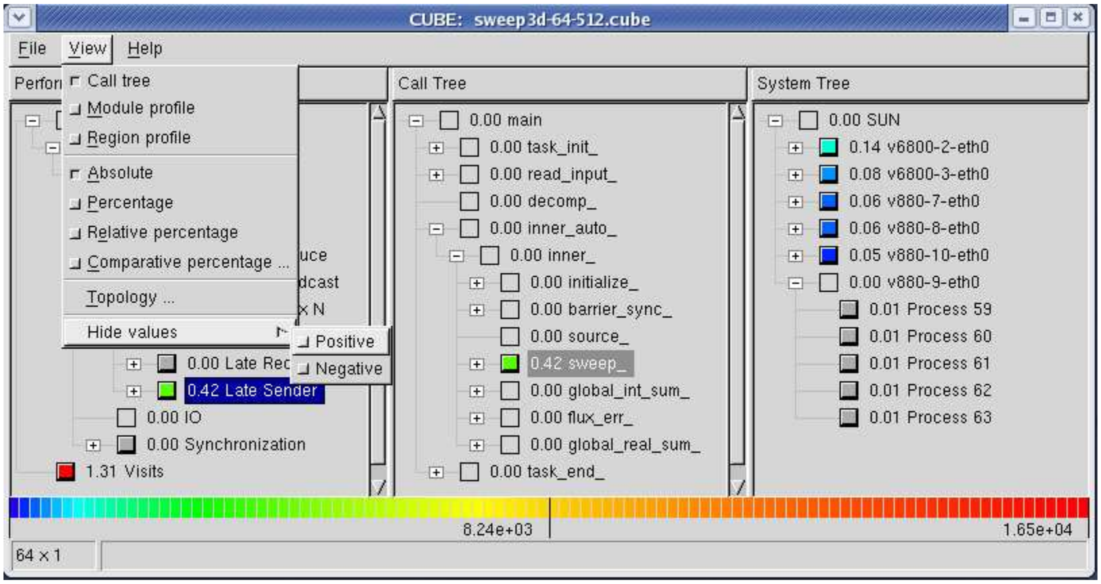

Figure 3: CUBE menu bar.

3.2.2 Menu Bar

The menu bar consists of three menus, a file menu, a view menu, and a help menu.

File

The file menu can be used to open and close a file and to exit CUBE.

View

The view menu (Figure 3) can be used to alter the way the program dimension is displayed, to change the number representation for the entire display, or to hide positive or negative values.

After opening a data set the middle panel shows the call tree of the program - unless the data set contains a flat profile. However, a user might wish to know which fraction of a metric can be attributed to a particular region regardless of from where it was called. In this case, the user can switch from the call-tree mode (default) to the module-profile mode or the region-profile mode (Figure 4). In the module-profile mode, the call-tree hierarchy is replaced with a source-code hierarchy consisting of three levels: module, region, and subregions. The subregions, if applicable, are displayed as a single child node labeled *subregions*. A *subregions* node represents all regions directly called from the region above. In this way, the user is able to see which fraction of a metric is associated with a region exclusively, that is, without its regions called from there. The region-profile mode is similar to the module-profile mode except that `modules` are not shown.

The severity can be displayed in four different ways: as an *absolute value* (default), a *percentage*, a *relative percentage*, or as a *comparative percentage*. The absolute value is the real value measured. When displaying a value as a percentage, the percentage refers to the value shown at the root of the metric tree when it is in collapsed state. However, both absolute mode and percentage mode have the disadvantage that values can become very small the more you go to the right, since aggregation occurs from right to left. To avoid this problem, the user can switch to relative percentages. Then, a percentage in the right or middle tree always refers to the selection in the neighbor to the left, that is, a percentage in the system dimension refers to the selection in the program dimension and a percentage in the program dimension refers to the selected metric dimension. In this mode the percentages in the middle and right tree always sum up to one hundred percent. Figure 4 shows a region profile with relative percentages. Furthermore, to facilitate the comparison of different experiments, users can choose the comparative percentage mode to display percentages relative to another data set. The comparative percentage mode is basically like the normal percentage mode except that the value equal to 100% is determined by another data set. Note that in the absolute mode, all values are displayed in scientific notation. To prevent cluttering the display, only the mantissa is shown at the nodes with the exponent displayed at the color legend.

If one or more virtual topologies have been defined in the CUBE file, the *Topology* menu item is enabled. Otherwise it is disabled. After selecting *Topology*, the Cartesian-selection dialog pops up if the CUBE file has multiple topologies. Through this dialog, users can choose a specific topology view to display in a topology window. Each topology is displayed in a separate window. Please refer to Section 3.3 for detailed information.

Finally, to help users distinguish between positive and negative values more easily, users can hide either positive or negative values.

Help

Currently, the help menu provides only an About dialog with release information.

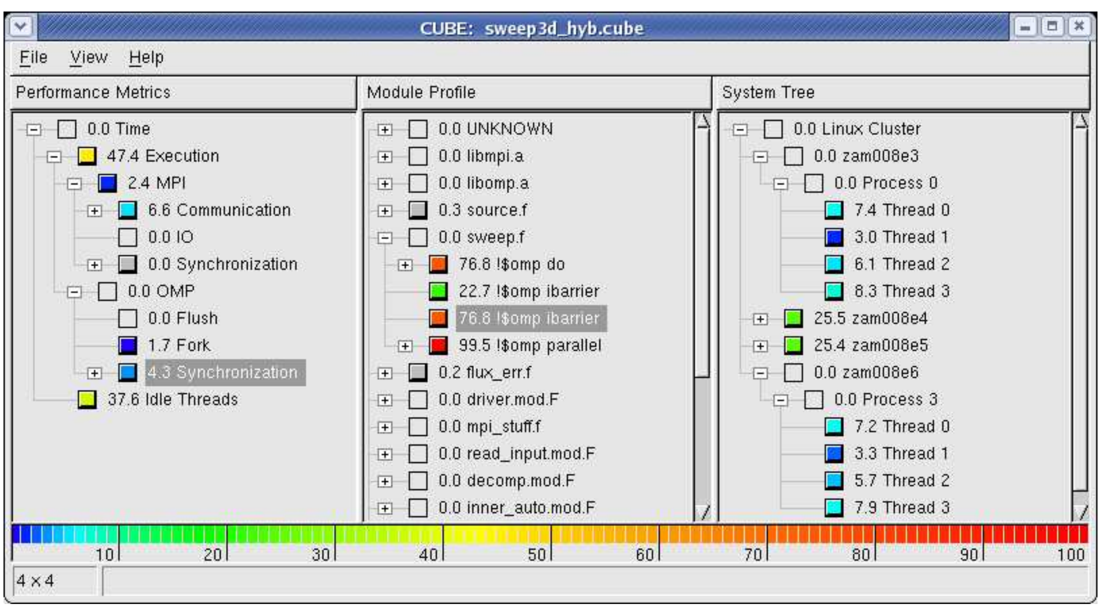

Figure 4: CUBE module profile.

3.2.3 Color Legend

The color is taken from a spectrum ranging from blue to red representing the whole range of possible values. To avoid an unnecessary distraction, insignificant values close to zero are displayed in dark gray. Exact zero values just have the background color. Depending on the severity representation, the color legend shows a numeric scale mapping colors onto values.

3.2.4 Status Bar

The first column showing $m \times n$ indicates that there are m processes and for each process there are at most n threads in the execution.

3.2.5 Context Menus

The metric and program dimensions provide a context menu that can be used to obtain specific information on each node. The context menu is accessible via the right mouse button. It displays all or a subset of the options described below.

The call tree has a context menu consisting of two levels. The first-level menu items are *Call site* and *Called region*. Choosing the *Call site* menu shows the information related to the call site, and choosing the *Called region* menu shows the information related to the region being called by the call site (i.e., the callee).

`Location`: Displays the source-code location of a program resource in textual form (i.e., at which line and in what module). In the module-profile and region-profile modes, it always refers to the location of its associated region. In the call-tree mode, a call-tree node is usually associated with two entities: a callsite and the region called by the callsite. By entering a specific level of the context menu: *Callsite* or *Called region*, users are able to check either the associated call site's or the called region's location. For the call site, it shows the call site's location where it has been called or its calling region's location if the line number of the call site is undefined. For the called region, it shows the location of the region being called by the call site.

`Source code`: Displays and highlights the source code of a program resource in the source code browser. In the module-profile and region-profile modes, it always shows and highlights the source code of its associated region. In the call-tree mode, since each call-tree node has a context menu of two levels, by choosing the *Call site* menu it displays and highlights the source code of the call site or the block of source code of the calling region. And by choosing the *Called region* menu it displays and highlights the block of code of the region being called by the call site. Note that not all data sets provide sufficient line-number information to show the correct section of the source code.

`Online description`: Both metrics and regions can be linked to an online description. For example, metrics might point to an online documentation explaining their semantics, or regions representing library functions might point to the corresponding library documentation.

`Info`: A brief description of metrics or regions supplied by the CUBE data set.

3.3 Topology Display

In many parallel applications, each process (or thread) communicates only with a limited number of processes. The parallel algorithm divides the application domain into smaller chunks known as sub domains. A process usually communicates with processes owning sub domains adjacent to its own. The mapping of data onto processes and the neighborhood relationship resulting from this mapping is called *virtual topology*. Many applications use one or more virtual topologies (Figure 5) specified as one-, two- or three- dimensional Cartesian grids. The CUBE topology display shows performance data mapped onto the Cartesian topology of the application. The corresponding grid is specified by two parameters: number of dimensions and size of each dimension

The display consists of a menu bar and the actual Cartesian grid. The Cartesian grid is presented by planes stacked on top of each other in a three dimensional projection. The number of planes depends on the number of dimensions in the grid. Each plane is divided into squares. The number of squares depends on the dimension size. Each square represents a system resource (e.g a process) of the application and has a coordinate associate with it.

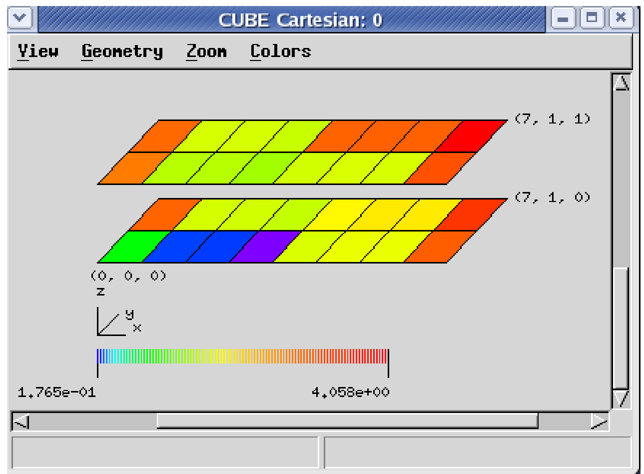

Figure 5: Topology display

The grid displays the severity of the selected metric in the selected call path for each system resource participating in the application's topology. The severity is represented as a color. A system resource might not be a part of the application's virtual topology or may have a zero value for a metric. Therefore, it is sometimes possible to have some uncolored squares in the grid picture.

3.3.1 Menu Bar

The menu bar consists of four menus: a view menu, a geometry menu, a zoom menu and, a colors menu.

`View`: The view menu can be used to choose one of the three possible orientations of the grid. The coordinate axes at the bottom of the picture indicate the direction of X, Y and Z dimensions in the three-dimensional space. In case of one- or two- dimensional grids, users are provided with only one orientation of the grid.

`Geometry`: Due to varying dimension sizes, planes in the grid might overlap with each other and the size of the squares might be too small to recognize their color. This may pose a problem for the user to view the topology information effectively. The geometry menu circumvents this problem by providing options to scale the picture in various ways. The *Angle* option helps the user to adjust the skew of the three-dimensional projection. The *Plane Distance* option helps to adjust the inter-plane distance. The *Plane Length* option helps users scale the area of each plane.

`Zoom`: The zoom menu can be used to zoom-in or zoom-out on the grid.

`Colors`: The colors menu can be used to modify the text color and the background color of the topology display. Finally, there are two resolution modes to choose from. The *Low Resolution* mode assigns colors to the squares according to the severity values shown in the system dimension. Often, these values have small variations from each other and do not help the user to study the relative distribution of severities across the grid. To exploit the entire spectrum of available colors and to enable the user to study the relative distribution of severities, a *High Resolution* mode is provided. This mode highlights the minute differences between severity values of the system resources. Severity values of zero are assigned the background color of the display. This mode has its own color legend showing the minimum and maximum values for the selected severities across the grid. These values can be absolute values, percentages, or relative percentages depending on the CUBE view mode.

4 Performance Algebra

As performance tuning of parallel applications usually involves multiple experiments to compare the effects of certain optimization strategies, CUBE offers a mechanism called *performance algebra* that can be used to merge, subtract, and average the data from different experiments and and view the results in the form of a single "derived" experiment. Using the same representation for derived experiments and original experiments provides access to the derived behavior based on familiar metaphors and tools in addition to an arbitrary and easy composition of operations. The algebra is an ideal tool to verify and locate performance improvements and degradations likewise. The algebra includes three operators *diff*, *merge*, and *mean* provided as command-line utilities which take two or more CUBE files as input and generate another CUBE file as output. The operations are closed in the sense that the operators can be applied to the results of previous operations. Note that although all operators are defined for any valid CUBE data sets, not all possible operations make actually sense. For example, whereas it can be very helpful to compare two versions of the same code, computing the difference between entirely different programs is unlikely to yield any useful results.

4.1 Difference

Changing a program can alter its performance behavior. Altering the performance behavior means that different results are achieved for different metrics. Some might increase while others might decrease. Some might rise in certain parts of the program only, while they drop off in other parts. Finding the reason for a gain or loss in overall performance often requires considering the performance change as a multidimensional structure. With CUBE's difference operator, a user can view this structure by computing the difference between two experiments and rendering the derived result experiment like an original one. The difference operator takes two experiments and computes a derived experiment whose severity function reflects the difference between the minuend's severity and the subtrahend's severity.

Figure 6 shows the difference between KOJAK [5] analysis results obtained from the original and an optimized version of a nano-particle simulation. Raised reliefs indicate performance improvements, and sunken reliefs indicate performance degradations. The figure indicates that a certain optimization was only partially successful because some of the wait states migrated to other locations in the program instead of disappearing.

Usage: `cube_diff <minuend> <subtrahend> [-o <output>]`

The default output file name is `diff.cube`.

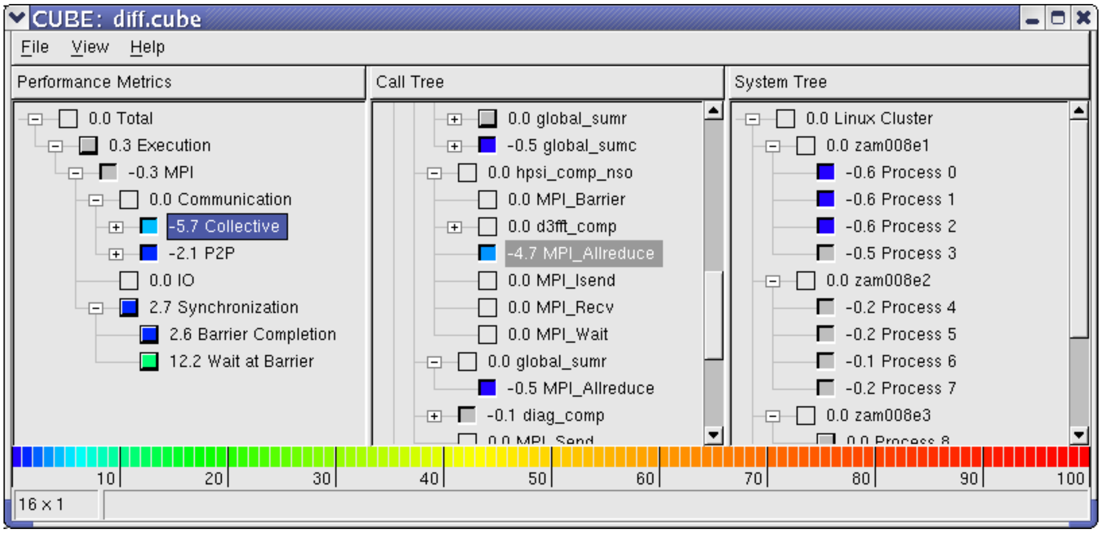

Figure 6: A derived experiment computed using `cube_diff`.

4.2 Merge

The merge operator's purpose is the integration of performance data from different sources. Often a certain combination of performance metrics cannot be measured during a single run. For example, certain combinations of hardware events cannot be counted simultaneously due to hardware resource limits. Or the combination of performance metrics requires using different monitoring tools that cannot be deployed during the same run. The merge operator takes two CUBE experiments with a different or overlapping set of metrics and yields a derived CUBE experiment with a joint set of metrics.

Usage: `cube_merge <op1> <op2> [-o <output>]`

The default output file name is `merge.cube`.

4.3 Mean

The mean operator is intended to smooth the effects of random errors introduced by unrelated system activity during an experiment or to summarize across a range of execution parameters. The user can conduct several experiments and create a single average experiment from the whole series. Different from the previous tow operators, which are binary operators, the mean operator takes an arbitrary number of arguments.

Usage: `cube_mean <op1> <op2> ...<opN> [-o <output>]`

The default output file name is `merge.cube`.

4.4 Implementation

The actions performed by the operators can be divided into two subtasks: integration of the performance space followed by the actual arithmetic operation.

4.4.1 Integration of the Performance Space

The integration of two or more performance spaces consists of three separate parts: merging the metric dimension, merging the program dimension, and merging the system dimension. Merging metric trees and call trees works very similar to the structural merge operator in [1]. While traversing from the roots to the leaves, CUBE tries to match up the nodes. Nodes that cannot be matched are separately included in the new performance space, whereas nodes that can be successfully matched become shared nodes, that is, they appear as a single node in the new space.

Merging the system dimension is slightly different. There are four levels with different meanings: machine, node, process, and thread. First, processes and threads are matched based on their application-level identifiers, for example, their global MPI rank and OpenMP thread number. Next, CUBE examines whether the partitioning of processes into nodes is compatible between the two operands. If compatible, CUBE copies the entire node and machine hierarchy including the corresponding process-node mapping of one of the operands into the result. Otherwise it collapses the machine and node level into a single machine and a single node.

4.4.2 Arithmetic Operation

After performance-space integration, a new severity function is computed whose domain is the integrated space. An element-wise operation is performed on the two input arrays. To be able to perform an element-wise operation, the operand's severity function is extended with respect to the integrated metadata so that it is defined for every tuple (metric, call path, thread) of the new metadata. This is done by assigning zero to previously undefined tuples. For example, a call path occurring in one metadata set might not occur in another. If this happens the resulting value for this call path will be set to zero in those experiments that didn't contain the call path before.

In the case of the difference or the mean operators, the element-wise operation is just a subtraction or arithmetic mean operation, respectively. In the case of the merge operator CUBE makes a simple case distinction. Recall that the purpose of doing the merge operation is to integrate performance experiments with different metrics. For example, one experiment counts floating point operations, and the other one counts cache misses, since we might not be able to count both of them simultaneously. So, if the metric is provided only by one experiment CUBE takes the data from that experiment. If it is provided by both experiments CUBE takes it from the first one (without loss of generality).

5 Tools

5.1 tau2cube

TAU is designed to provide a framework for integrating program and performance analysis tools and components. Using the tool of `tau2cube`, TAU profiles are able to be converted to the CUBE format.

Usage: `tau2cube [<tau-profile-dir>] [-o <output>]`

The first parameter is the TAU profile directory. The second parameter is the name of the output CUBE file. The default for <tau-profile-dir> is the current directory and the default output file name is `a.cube`.

Limitations:

- Converts only flat, two-level (one level more than flat), or full call path profiles (all callers up to main).
- The main function must be included and every other function must be called from within main. Static initializers outside the main function are not supported.

6 Creating CUBE Files

The CUBE data format in an XML instance [6]. The corresponding XMLSchema specification [7] can be found in `doc/cube.xsd` in the CUBE distribution. The CUBE library provides an interface to create CUBE files. It is a simple class interface and includes only a few methods. This section first describes the CUBE API and then presents a simple C++ program as an example of how to use it.

6.1 CUBE API

The class interface defines a `class Cube`. The class provides a default constructor and sixteen methods. The methods are divided into four groups. The first three groups are used to define the three dimensions of the performance space and the last group is used to enter the actual data. In addition, an output operator << to write the data to a file is provided.

The methods used to create the different entities of the performance space always return a `const` object pointer which can be used for further reference.

6.1.1 Metric Dimension

This group refers to the metric dimension of the performance space. It consist of a single method used to build metric trees. Each node in the metric tree represents a performance metric. Metrics have different units of measurement. The unit can be either "`sec`" (i.e., seconds) for time based metrics, such as execution time, or "`occ`" (i.e., occurrences) for event-based metrics, such as floating-point operations. During the establishment of a metric tree, a child metric is usually more specific than its parent, and both of them have same unit of measurement. Thus, a child performance metric has to be a subset of its parent metric (e.g., system time is a subset of execution time).

```
const Metric* def_met   (string name, string uom, string url,
                         string descr, const Metric* parent);
```

Returns a metric with name `name` and description `descr`. `uom` specifies the unit of measurement, which is either "`sec`" for seconds or "`occ`" for number of occurrences. `parent` is a previously created metric which will be the new metric's parent. To define a root node, use `NULL` instead. `url` is a link to an HTML page describing the new metric in detail. If you want to mirror the page at several locations, you can use the macro `@mirror@` as a prefix, which will be replaced by an available mirror defined using `def_mirror()` (see Section 6.1.6).

6.1.2 Program Dimension

This group refers to the program dimension of the performance space. The entities presented in this dimension are *module*, *region*, *call site*, and *call-tree node* (i.e., call paths). A module is a source file, which can contain several code regions. A region can be a function, a loop, or a basic block. Each region can have multiple call sites from which the control flow of the program enters a new region. Although we use the term call site here, any place that causes the program to enter a new region can be represented as a call site, including loop entries. Correspondingly, the region entered from a call site is called *callee*, which might as well be a loop. Every call-tree node points to a call site. The actual call path represented by a call-tree node can be derived by following all the call sites starting at the root node and ending at the particular node of interest. Therefore, before defining a call-tree node, the necessary call sites, callees, and modules have to be defined. The user can chose among three ways of defining the program dimension:

1. Call tree with line numbers
2. Call tree without line numbers
3. Flat profile

A call tree with line numbers is defined as a tree whose nodes point to call sites. A call tree without line numbers is defined as a tree whose nodes point to regions (i.e., the callees). A flat profile is simply defined as a set of regions, that is, no tree has to be defined.

```
const Module* def_module  (string name);
```

Returns a new module with module name `name`, which can be either a complete path or a file name.

```
const Region* def_region  (string name, long begln, long endln,
                            string url, string descr,
                            const Module* mod);
```

Returns a new region with region name `name` and description `descr`. The region is located in the module `mod` and exists from line `begln` to line `endln`. `url` is a link to an HTML page describing the new metric in detail. If you want to mirror the page at several locations, you can use the macro `@mirror@` as a prefix, which will be replaced by an available mirror defined using `def_mirror()` (see Section 6.1.6).

```
const Callsite* def_csite  (const Module* mod, int line,
                             const Region* callee);
```

Returns a new a call site located at the line `line` of the module `mod`. The call site calls the callee `callee` (i.e., a previously defined region).

```
const Cnode* def_cnode  (const Callsite* csite,
                          const Cnode* parent);
```

Returns a new call-tree node representing a call from call site `csite`. `parent` is a previously created call-tree node which will be the new one's parent. To define a root node, use `NULL` instead. This method is used to create a call tree with line numbers.

```
const Cnode* def_cnode  (const Region* region,
                         const Cnode* parent);
```

Defines a new call-tree node representing a call to the region `region`. `parent` is a previously created call-tree node which will be the new one's parent. To define a root node, use `NULL` instead. Note that different from the previous `def_cnode()`, this method is used to create a call-tree without line numbers where each call-tree node points to a region, instead of a call site.

To define a call tree with line numbers use `def_csite()` and `def_cnode(const Callsite*...)`. To define a call tree without line numbers use `def_cnode(const Region*...)`. To create a flat profile use neither one - just defining a set of regions will be sufficient.

6.1.3 System Dimension

This group refers to the system dimension of the performance space. It reflects the system resources on which the program is using at runtime. The entities present in this dimension are *machine*, *node*, *process*, and *thread*, which populate four levels of the system hierarchy in the given order. That is, the first level consists of machines, the second level of nodes, and so on. Finally, the last (i.e., leaf) level is populated only by threads. The system tree is built in a top-down way starting with a machine. Note that even if every process has only one thread, users still need to define the thread level.

```
const Machine* def_mach  (string name);
```

Returns a new machine with the name `name`.

```
const Node* def_node  (string name, const Machine* mach);
```

Returns a new (SMP) node which has the name `name` and which belongs to the machine `mach`.

```
const Process* def_proc  (string name, int rank,
                          const Node* node);
```

Returns a new process which has the name `name` and the rank `rank`. The rank is a number from $0-(n-1)$, where n is the total number of processes. MPI applications may use the rank in `MPI_COMM_WORLD`. The process runs on the node `node`.

```
const Thread* def_thrd  (string name, int rank,
                         const Process* proc);
```

Defines a new thread which has the name `name` and the rank `rank`. The rank is a number from $0-(n-1)$, where n is the total number of threads spawned by a process. OpenMP applications may use the OpenMP thread number. The thread belongs to the process `proc`.

6.1.4 Virtual Topologies

Virtual topologies are used to describe adjacency relationships among machines, SMP nodes, processes or threads. A topology usually consists of a single class of entities such as threads or processes. The CUBE API provides a set of functions to create Cartesian topologies and to define the machine/SMP node/process/thread mappings onto coordinates. Note that the definition of virtual topologies is optional.

```
const Cartesian* def_cart  (long ndims, const vector<long>& dimv,
                             const vector<bool>& periodv);
```

Defines a new Cartesian topology. `ndims` and `dimv` specify the number of dimensions and the size of each dimension. `periodv` specifies the periodicity for each dimension.

```
void def_coords  (const Cartesian* cart, const Location* loc,
                  const vector<long>& coordv);
```

Maps a specific location onto a Cartesian coordinate. The location `loc` may be a machine, SMP node, process or a thread. It is not recommended to map a mixed set of entities onto one topology (e.g., machines and threads are located in the same topology). The parameter of `cart` has been defined by the above `def_cart()` method.

6.1.5 Severity Mapping

After the establishment of performance space, users can assign severity values to points of the space. Each point is identified by a tuple `(met, cnode, thrd)`. The value should be inclusive with respect to the metric, but exclusive with respect to the call-tree node, that is it should not cover its children. Taking Figure 2 as an example, this means that if the value refers to *main* then it should not include *main-foo* or *main-bar*. The default severity value for the data points left undefined is zero. Thus, users only need to define non-zero data points.

```
void set_sev  (const Metric* met, const Cnode* cnode,
               const Thread* thrd, double value);
```

Assigns the value `value` to the point `(met, cnode, thrd)`.

```
void add_sev  (const Metric* met, const Cnode* cnode,
               const Thread* thrd, double value);
```

Adds the value `value` to the present value at point `(met, cnode, thrd)`.

The previous two methods `set_sev()` and `add_sev()` are intended to be used when the program dimension contains a call tree and not a flat profile. As the flat profile does not require the definition of call-tree nodes, the following two functions should be used instead:

```
void set_sev  (const Metric* met, const Region* region,
               const Thread* thrd, double value);
```

Assigns the value `value` to the point `(met, region, thrd)`.

```
void add_sev  (const Metric* met, const Region* region,
               const Thread* thrd, double value);
```

Adds the value `value` to the present value at point `(met, region, thrd)`.

6.1.6 Miscellaneous

Often users may want to define some information related to the CUBE file itself, such as the creation date, experiment platform, and so on. For this purpose, CUBE allows the definition of arbitrary attributes in every CUBE data set. An attribute is simply a key-value pair and can be defined using the following method:

`void def_attr (string key, string value);`

Assigns the value `value` to the attribute `key`.

There is one predefined attribute CUBE_CT_AGGR with values MAX and SUM to stipulate the aggregation mode applied in the presence of multiple call trees (Section 3.1). If this attribute is defined CUBE will use the specified mode and suppress the input dialog.

CUBE allows using multiple mirrors for the online documentation associated with metrics and regions. The `url` expression supplied as an argument for `def_metric()` and `def_region()` can contain a prefix `@mirror@`. When the online documentation is accessed, CUBE can substitute all mirrors defined for the prefix until a valid one has been found. If no valid online mirror can be found, CUBE will substitute the `./doc` directory of the installation path for `@mirror@`.

`void def_mirror (string mirror);`

Defines the mirror `mirror` as potential substitution for the URL prefix `@mirror@`.

6.2 Typical Usage

A simple C++ program is given to demonstrate how to use the CUBE write interface. Figure 7 shows the corresponding CUBE display. The source code of the target application is provided in Figure 8.

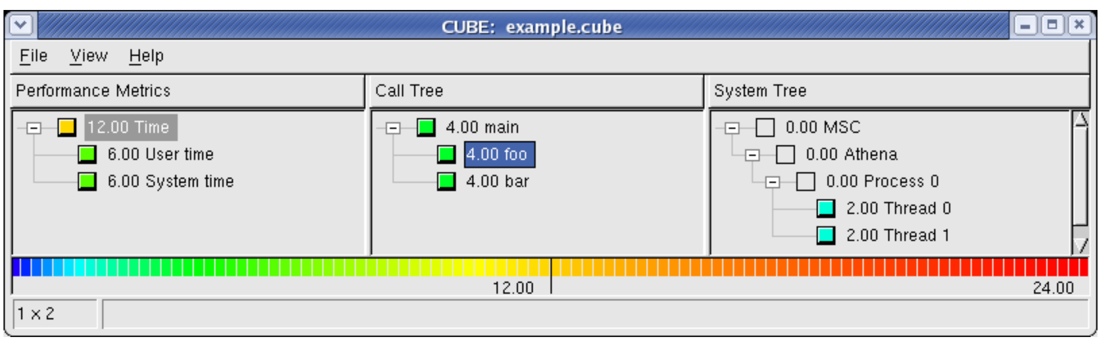

Figure 7: Display of `example.cube`

```
// A C++ example using CUBE write interface
int main(int argc, char* argv[]) {
  // Declarations (All const class pointers)

    ...

  Cube cube;

  // specify mirrors (optional)
```

```
1         void foo() {
              ...
10        }
11        void bar() {
              ...
20        }
21        int main(int argc, char* argv) {
              ...
60          foo();
              ...
80          bar();
              ...
100       }
```

Figure 8: Target-application source code `example.c`

```
cube.def_mirror("http://icl.cs.utk.edu/software/kojak/");
cube.def_mirror("http://www.fz-juelich.de/zam/kojak/");

// specify information related to the file (optional)
cube.def_attr("experiment time", "November 1st, 2004");
cube.def_attr("description", "a simple example");

// build metric tree
met0 = cube.def_met("Time", "sec",
                    "@mirror@patterns-2.1.html#execution",
                    "root node", NULL); // using mirror
met1 = cube.def_met("User time", "sec",
                    "http://www.cs.utk.edu/usr.html",
                    "2nd level", met0); // withoug using mirror
met2 = cube.def_met("System time", "sec",
                     "http://www.cs.utk.edu/sys.html",
                    "2nd level", met0); // withoug using mirror

// build a call tree with line numbers
mod    = cube.def_module("/ICL/CUBE/example.c");
regn0  = cube.def_region("main", 21, 100, "", "1st level", mod);
regn1  = cube.def_region("foo", 1, 10, "", "2nd level", mod);
regn2  = cube.def_region("bar", 11, 20, "", "2nd level", mod);

// When creating flat profiles, you do not need
// define call sites and call-tree nodes.
csite0 = cube.def_csite(mod, 21, regn0);
csite1 = cube.def_csite(mod, 60, regn1);
csite2 = cube.def_csite(mod, 80, regn2);
cnode0 = cube.def_cnode(csite0, NULL);
cnode1 = cube.def_cnode(csite1, cnode0);
cnode2 = cube.def_cnode(csite2, cnode0);
/* If creating call trees without line numbers,
   put a region as the 1st argument in the
```

```
     above def_cnode()'s and don't define csites */

  // build system tree
  mach  = cube.def_mach("msc");
  node  = cube.def_node("athena", mach);
  proc  = cube.def_proc("Process 0", 0, node);
  thrd0 = cube.def_thrd("Thread", 0, proc);
  thrd1 = cube.def_thrd("Thread", 1, proc);

  // build a 2D Cartesian topology (a 5x5 grid)
  int ndims = 2;
  vector<long> dimv;
  vector<bool> periodv;
  dimv.push_back(5);
  dimv.push_back(5);
  const Cartesian* cart = cube.def_cart(ndims, dimv, periodv);
  vector<long> coord0, coord1;
  coord0.push_back(0);
  coord0.push_back(0);
  coord1.push_back(3);
  coord1.push_back(3);
  // map the two threads onto the above 2 coordinates.
  cube.def_coords(cart, thrd0, coord0);
  cube.def_coords(cart, thrd1, coord1);

  // severity mapping
  cube.set_sev(met0, cnode0, thrd0, 4);
  cube.set_sev(met0, cnode0, thrd1, 4);
  cube.set_sev(met0, cnode1, thrd0, 4);
  cube.set_sev(met0, cnode1, thrd1, 4);
  cube.set_sev(met0, cnode2, thrd0, 4);
  cube.set_sev(met0, cnode2, thrd1, 4);
  cube.set_sev(met1, cnode0, thrd0, 1);
  cube.set_sev(met1, cnode0, thrd1, 1);
  cube.set_sev(met1, cnode1, thrd0, 1);
  cube.set_sev(met1, cnode1, thrd1, 1);
  cube.set_sev(met1, cnode2, thrd0, 1);
  cube.set_sev(met1, cnode2, thrd1, 1);
  cube.set_sev(met2, cnode0, thrd0, 1);
  cube.set_sev(met2, cnode0, thrd1, 1);
  cube.set_sev(met2, cnode1, thrd0, 1);
  cube.set_sev(met2, cnode1, thrd1, 1);
  cube.set_sev(met2, cnode2, thrd0, 1);
  cube.set_sev(met2, cnode2, thrd1, 1);

  // when creating a flat profile, put a region as the 2nd argument in
  // the above set_sev() calls

  // write output to a file
  ofstream out;
  out.open("example.cube");
  out << cube;
}
```

References

[1] K. L. Karavanic and B. Miller. A Framework for Multi-Execution Performance Tuning. *Parallel and Distributed Computing Practices*, 4(3), September 2001. Special Issue on Monitoring Systems and Tool Interoperability.

[2] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org`.

[3] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface - Version 2.0*, November 2000. `http://www.openmp.org`.

[4] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An Algebra for Cross-Experiment Performance Analysis. In *Proc. of ICPP 2004*, pages 63–72, Montreal, Canada, August 2004.

[5] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue "Evolutions in parallel distributed and network-based processing".

[6] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000. `http://www.w3.org/TR/REC-xml`.

[7] World Wide Web Consortium. *XML Schema Part 0, 1, 2*, May 2001. `http://www.w3.org/XML/Schema#dev`.