JÜLICH
FORSCHUNGSZENTRUM

CUBE 4.3.2 – Cube Tools Developer Guide

Commented example of a CUBE tool

June 12, 2015

The Scalasca Development Team
scalasca@fz-juelich.de

Copyright

Copyright © 1998–2015 Forschungszentrum Jülich GmbH, Germany

Copyright © 2009–2015 German Research School for Simulation Sciences GmbH, Jülich/Aachen, Germany

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the names of Forschungszentrum Jülich GmbH or German Research School for Simulation Sciences GmbH, Jülich/Aachen, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

1 Makefile for provided examples

1.1 Quick info about makefile.

Here we provide a small example of a makefile, which is used to compile and build examples delivered with CUBE. Similar makefiles can be used by developers to compile and build own CUBE tools.

1.2 Commented source

First we specify the installation path of CUBE and its "cube-config" script. This script delivers correct flags for compiling and linking, paths to the CUBE tools and GUI. (besides of another useful technical information)

```
CUBE_DIR = /path/CubeInstall
CUBE_CONFIG = $(CUBE_DIR)/bin/cube-config
```

Additionally we specify CPPFLAGS and LDFLAGS to compile and link examples.

```
CPPFLAGS = $(shell $(CUBE_CONFIG) --cube-cxxflags)
CFLAGS = $(shell $(CUBE_CONFIG) --cubew-cxxflags)
CLDFLAGS = $(shell $(CUBE_CONFIG) --cubew-ldflags)
CPPLDFLAGS = $(shell $(CUBE_CONFIG) --cube-ldflags)
```

Here a compiler is selected to compile and build the example.

```
# GNU COMPILER
CXX = g++
CC = gcc -std=c99
MPICXX= mpicxx
```

We define explicit suffixes for an executable file, created from C source, from c++ source and an MPI executable. If one develops a tool, which is using MPI, it is useful (sometimes) to define a special suffix for automatic compilation.

```
.SUFFIXES: .c .o .cpp .c.exe .cpp.exe .c.o .cpp.o .mpi.o .mpi.cpp
.PHONY: all  clean
```

Object files of examples and their targets

```
# Object files
OBJS =  cube_example.cpp.o \
        cubew_example.c.o

TARGET = cube_example.cpp.exe \
         cubew_example.c.exe
```

Automatic rule for the compilation of every single C++ source into .o file and for building targets.

```
%.cpp.o : %.cpp
        $(CXX) -c $< -o $@ $(CPPFLAGS)

%.cpp.exe : %.cpp.o
        $(CXX)  $< -o $@ $(CPPLDFLAGS)
```

Automatic rule for the compilation of every single C++ with MPI source into .o file and for building targets.

```
%.mpi.o : %.mpi.cpp
        $(MPICXX) -c $< -o $@ $(CPPFLAGS) $(CFLAGS)

%.mpi.exe : %.mpi.o
        $(MPICXX) $< -o $@  $(CLDFLAGS)
```

Automatic rule for the compilation of every single C source into .o file and for building targets.

```
%.c.o : %.c
        $(CC) -c $< -o $@ $(CFLAGS)

%.c.exe : %.c.o
        $(CC)  $< -o $@ $(CLDFLAGS)


#------------------------------------------------------------------------------
# Rules
#------------------------------------------------------------------------------

all: $(TARGET)
```

2 Examples of using C++ library

Present example shows in several short steps the main idea of creating a cube file using C++ library and obtaining different values from this cube file.

2.1 Commented source

Include standard c++ header

```
....
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
```

Include CUBE headers. Notice, that all C++ headers of Cube framework have a prefix CubeXXX.h.

```
#include "Cube.h"
#include "CubeMetric.h"
#include "CubeCnode.h"
#include "CubeProcess.h"
#include "CubeNode.h"
#include "CubeThread.h"
#include "CubeCartesian.h"
```

Define namespaces, where we do operate, standard templates library and cube namespace.

```
using namespace std;
using namespace cube;
```

Start main and declare needed variables. We want to create three metrics and some regions.

```
int main(int argc, char* argv[]) {
  Metric *met0, *met1, *met2;
  Region *regn0, *regn1, *regn2;
  Cnode  *cnode0, *cnode1, *cnode2;
  Machine* mach;
  Node* node;
  Process* proc0, *proc1;
  Thread *thrd0, *thrd1;
```

Enclose everything in a *try-catch* block, because the cube library throws exceptions, if something goes wrong.

```
try
{
```

Create cube object. This means we create a new empty Cube.

```
Cube cube;
```

Specify general properties of the cube object.

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

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



cube.set_statistic_name("statisticstat");
cube.set_metrics_title("Metrics");
cube.set_calltree_title("Calltree");
cube.set_systemtree_title("Systemtree");
```

Now we start to define dimensions of the cube.

First we define the `metric` dimension. Notice, that metrics build a tree and parents have to be declared before their children.

Every metric can be of one of two kinds: *inclusive* or *exclusive*.

If one omits `CUBE_INCLUSIVE` or `CUBE_EXNCLUSIVE`, cube automatically creates an *exclusive* metric. One cannot change this type later.

Every metric needs a display name, an unique name, type of values (INTEGER, DOUBLE, MAXDOUBLE, MINDOUBLE, others), units of measurement, value (usually empty string), URL, where one can find the documentation about this metric, description and its parent in the metric tree.

The cube returns a pointer on struct *cube_metric*, which has to be used for saving or reading values from the cube.

```
// Build metric tree
met0 = cube.def_met("Time", "Uniq_name1", "INTEGER", "sec", "",
                    "@mirror@patterns-2.1.html#execution",
                    "root node", NULL, CUBE_INCLUSIVE); // using mirror
met1 = cube.def_met("User time", "Uniq_name2", "INTEGER", "sec", "",
                    "http://www.cs.utk.edu/usr.html",
                     "2nd level", met0); // without using mirror
met2 = cube.def_met("System time", "Uniq_name3", "INTEGER", "sec", "",
                     "http://www.cs.utk.edu/sys.html",
                     "2nd level", met0); // without using mirror
```

Then we define another dimension, the "call tree" dimension. This dimension gets defined in a two-step way:

1. One defines a list of regions in the instrumented source code;
2. One builds a call tree with the regions defined above;

Then we define the `call tree` dimension. This dimension gets defined in a two-step way:

1. One defines a list of regions in the instrumented source code;
2. One builds a call tree with regions defined in the previous step.

First one defines regions.

Every region has a name, start and end line, URL with the documentation of the region, description and source file (module). Regions build a list, therefore no "parent-child" relation is given.

The cube returns a pointer on Object *Region*, which can be used later for the calculations, visualization or access to the data.

```
// Build call tree
string mod    = "/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);
```

Then one defines an actual dimension, the `call tree` dimension.

The call tree consists of so called CNODEs. Cnode stands for "call path".

Every cnode gets as a parameter a region, source file (module), its id and parent cnode (caller).

Parent cnodes have to be defined before their children. Region might be entered from different places in the program, therefore different cnodes might have same region as a parameter.

```
cnode0 = cube.def_cnode(regn0, mod, 21, NULL);
cnode1 = cube.def_cnode(regn1, mod, 60, cnode0);
cnode2 = cube.def_cnode(regn2, mod, 80, cnode0);
```

The last dimension is the `system tree` dimension. Currently CUBE defines the system dimension with the fixed hierarchy: `MACHINE` $\rightarrow$ `NODES` $\rightarrow$ `PROCESSES` $\rightarrow$ `THREADS`

It leads to the fixed sequence of calls in the system dimension definition:

1. First one creates a root for the system dimension : *Machine*. Machine has a name and description.
2. Machine consists of *Node*s. Every *Node* has a name and a *Machine* as a parent.
3. On every *Node* run several *cube_process*es (as many cores are available). *Process* has a name, MPI rank and *Node* as a parent.
4. Every *Process* spawns several (one or more) *Thread*s (OMP, Pthreads, Java Threads). *Thread* has a name, its rank and *Process* as a parent.

The cube returns a pointer on *CubeMachine*, *Node*, *Process* or *Thread*, which has to be used later to define further level in the system tree or to access the data in the cube.

```
// Build location tree
mach  = cube.def_mach("MSC", "");
node  = cube.def_node("Athena", mach);
proc0 = cube.def_proc("Process 0", 0, node);
proc1 = cube.def_proc("Process 1", 1, node);
thrd0 = cube.def_thrd("Thread 0", 0, proc0);
thrd1 = cube.def_thrd("Thread 1", 1, proc1);
```

After the dimensions are defined, one fills the cube object with the data. Every data value is specified by three "coordinates": (*Metric*, *Cnode*, *Thread*)

Note, that *Machine*, *Node* and *Process* are not a "coordinate". They are used only to build up the physical construction of the machine.

C++ cube library allows random access to the data, therefore no specific restrictions about sequence of calls are made.

```
cube.set_sev( met0, cnode0, thrd0, 12. );
cube.set_sev( met0, cnode0, thrd1, 11. );
cube.set_sev( met0, cnode1, thrd0, 5. );
cube.set_sev( met0, cnode1, thrd1, 6. );
cube.set_sev( met0, cnode2, thrd0, 4.2 );
cube.set_sev( met0, cnode2, thrd1, 3.5 );

cube.set_sev( met1, cnode0, thrd0, 4. );
cube.set_sev( met1, cnode0, thrd1, 4. );
cube.set_sev( met1, cnode1, thrd0, 3. );
cube.set_sev( met1, cnode1, thrd1, 3.2 );
cube.set_sev( met1, cnode2, thrd0, 0.2 );
cube.set_sev( met1, cnode2, thrd1, 0.5 );

cube.set_sev( met2, cnode0, thrd0, 2. );
cube.set_sev( met2, cnode0, thrd1, 2.4 );
cube.set_sev( met2, cnode1, thrd0, 1.2 );
cube.set_sev( met2, cnode1, thrd1, 1.2 );
cube.set_sev( met2, cnode2, thrd0, 0.01 );
cube.set_sev( met2, cnode2, thrd1, 0.03 );

cube.set_sev( met3, cnode0, thrd0, 10 );
cube.set_sev( met3, cnode0, thrd1, 20 );
cube.set_sev( met3, cnode1, thrd0, 800 );
cube.set_sev( met3, cnode1, thrd1, 700 );
cube.set_sev( met3, cnode2, thrd0, 9300 );
cube.set_sev( met3, cnode2, thrd1, 9500 );
```

There are different calls to get a value from cube:

1. `double value1 = cube.get_sev(met0, cnode0, thrd0);`

   Get a double representation of a single value for a given metric, call path and thread. This call is backward compatible to cube3 and delivers an exclusive value along the call tree.

2.
   ```
   double value2 =
      cube.get_sev(met0,   cube::CUBE_CALCULATE_INCLUSIVE,
                   cnode0, cube::CUBE_CALCULATE_INCLUSIVE,
                   thrd0 , cube::CUBE_CALCULATE_INCLUSIVE);
   ```

   Get a double representation of a single value for a given metric, call path and thread. Here one specifies kind of value along every three. Notice, that this time it delivers inclusive value along the call tree (Metric values are naturally inclusive and threads are leaves in the current model of the system tree)

3.
   ```
   double value3 =
      cube.get_sev(met0,   cube::CUBE_CALCULATE_INCLUSIVE,
                   cnode0, cube::CUBE_CALCULATE_INCLUSIVE,
                   node , cube::CUBE_CALCULATE_INCLUSIVE);
   ```

   Get a double representation of a single value for a given metric, call path and whole node. It means, aggregation over all threads of all processes of this node is made.

4.
   ```
   double value4 =
      cube.get_sev(met0,   cube::CUBE_CALCULATE_INCLUSIVE,
                   cnode0, cube::CUBE_CALCULATE_INCLUSIVE
                   );
   ```

   Get a double representation of a single value for a given metric, call path, aggregated over whole system tree using algebra of the metric value.

5.
   ```
   double value5 =
      cube.get_sev(met0,   cube::CUBE_CALCULATE_EXCLUSIVE
                   );
   ```

   Get a double representation of a single value for a given metric (exclusive along metric tree), aggregated over the whole call tree and system tree using algebra of the metric value.

There are several another additional calls of the same type. Please see documentation (Cube library source).

CUBE can carry a set of so called "topologies": mappings `THREAD` $\rightarrow$ (x, y, z, ...)

Then GUI visualize every value (*Metric*, *Cnode*, *Thread*) for the selected metric and cnode as a 1D, 2D or 3D set of points with the different colors.

First one specifies a number of dimensions (any number is supported, currently shown are only the first three), a vector with the sizes in every dimension and its periodicity and creates an object of class *Cartesian*

```
// building a topology
// create 1st cartesian.
int ndims = 2;
vector<long> dimv;
vector<bool> periodv;
for (int i = 0; i < ndims; i++) {
  dimv.push_back(5);
  if (i % 2 == 0)
    periodv.push_back(true);
  else
    periodv.push_back(false);
}

Cartesian* cart = cube.def_cart(ndims, dimv, periodv);
```

The coordinates one defines like a vector and creates a mapping.

```
if (cart != NULL)
{
  vector<long> p[2];
  p[0].push_back(0);
  p[0].push_back(0);
  p[1].push_back(2);
  p[1].push_back(2);
  cube.def_coords(cart, thrd1, p[0]);
}
```

One can define names for every dimension.

```
      vector<string> dim_names;
      dim_names.push_back( "X" );
      dim_names.push_back( "Y" );
      cart->set_namedims( dim_names );
```

In the same way one can create any number of topologies. They are shown in the GUI.

```
      ndims = 2;
      vector<long> dimv2;
      vector<bool> periodv2;
      for ( int i = 0; i < ndims; i++ )
      {
          dimv2.push_back( 3 );
          if ( i % 2 == 0 )
          {
              periodv2.push_back( true );
          }
```

```
            else
            {
                periodv2.push_back( false );
            }
        }


        Cartesian* cart2 = cube.def_cart( ndims, dimv2, periodv2 );
        cart2->set_name( "Application topology 2" );
        if ( cart2 != NULL )
        {
            vector<long> p2[ 2 ];
            p2[ 0 ].push_back( 0 );
            p2[ 0 ].push_back( 1 );
            p2[ 1 ].push_back( 1 );
            p2[ 1 ].push_back( 0 );
            cube.def_coords( cart2, thrd0, p2[ 0 ] );
            cube.def_coords( cart2, thrd1, p2[ 1 ] );
        }
        cart2->set_dim_name( 0, "Dimension 1" );
        cart2->set_dim_name( 1, "Dimension 2" );
```

To save cube file on disk, one calls the following method. Extension ".cubex" will be added automatically. There is a corresponding method `openCubeReport(...);`

```
        cube.writeCubeReport( "cube-example" );
```

There are some routines for the conversion from cube3 to cube4 called.

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


  // Read it (example.cube) in and write it to another file (example2.cube)
  ifstream in("example.cube");
  Cube cube2;

  in >> cube2;
  ofstream out2;
  out2.open("example2.cube");

  out2 << cube2;
```

Here we catch all exceptions, thrown by the cube. We print the exception message and finish the program.

```
  } catch(RuntimeError e)
```

```
    {
        cout << "Error: " << e.get_msg() << endl;
    }

}
```

3 Examples of using CUBE c-writer library

Present example shows in several short steps the main idea of creating a cube file using C writer library.

In this example we do not show the optimization, which is needed to prevent unnecessary `seek`s while writing.

3.1 Commented source

Include standard c header

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
```

Include CUBE headers.

Notice, that CUBE4 c-writer headers got an prefix `cubew_XXX.h`

```
#include "cubew_cube.h"
```

Start main and define a name of the cube file. Extension ".cubex" will be append automatically.

```
int main(int argc, char* argv[])
{

    char cubefile[12] = "simple-cube";
```

Create the structure of the cube. CUBE_MASTER defines, that in parallel MPI environment this cube (usually rank 0) writes all parts of the cube (anchor, indexes and data). The last argument is ignored in current version of CUBE c-writer.

```
cube_t* cube=cube_create("example", CUBE_MASTER, CUBE_FALSE);
if (!cube) {
    fprintf(stderr,"cube_create failed!\n");
    exit(1);
}
```

Specify general properties of cube object.

```
cube_def_mirror(cube, "http://icl.cs.utk.edu/software/kojak/");
cube_def_mirror(cube, "http://www.fz-juelich.de/jsc/kojak/");
cube_def_attr(cube, "description", "a simple example");
cube_def_attr(cube, "experiment time", "November 1st, 2004");

cube_set_statistic_name(cube, "mystat");
```

Now we start to define the dimensions of the cube.

First we define `metric` dimension. Notice, that metrics build a tree and parents have to be declared before their children.

Every metric can be of two kinds: *inclusive* or *exclusive*.

Every metric needs a display name, an unique name, type of values (INTEGER, DOUBLE, MAXDOUBLE, MINDOUBLE, others), units of measurement, value (usually empty string), URL, where one can find the documentation about this metric, description and its parent in the metric tree.

The cube returns a pointer on structure *cube_metric*, which has to be used for saving or reading values from the cube.

```
cube_metric *met0, *met1, *met2;
met0 = cube_def_met(cube, "Time", "Uniq_name1", "FLOAT", "sec", "",
            "@mirror@patterns-2.1.html#execution",
            "root node", NULL, CUBE_METRIC_EXCLUSIVE);
met1 = cube_def_met(cube, "User time", "Uniq_name2", "FLOAT", "sec", "",
            "http://www.cs.utk.edu/usr.html",
            "2nd level", met0, CUBE_METRIC_INCLUSIVE);
met2 = cube_def_met(cube, "System time", "Uniq_name3", "INTEGER", "sec", "",
            "http://www.cs.utk.edu/sys.html",
            "2nd level", met0, CUBE_METRIC_EXCLUSIVE);
```

Then we define the `calltree` dimension. This dimension gets defined in a two-step way:

1. One defines a list of regions in the instrumented source code;
2. One builds a call tree with the regions defined in the previous step.

First one defines the regions.

Every region has a name, start and end line, URL with the documentation of the region, description and source file (module). Regions build a list, therefore no "parent-child" relation is given.

The cube returns a pointer on structure *cube_region*, which can be used later for the calculations, visualization or access to the data.

```
char* mod = "/ICL/CUBE/example.c";
cube_region *regn0, *regn1, *regn2;
regn0 = cube_def_region(cube, "main", 21, 100, "", "1st level", mod);
regn1 = cube_def_region(cube, "<<init>>foo", 1, 10, "", "2nd level", mod);
regn2 = cube_def_region(cube, "<<loop>>bar", 11, 20, "", "2nd level", mod);
```

Then one defines an actual dimension, the `call tree` dimension.

Call tree consists of so called CNODEs. Cnode stands for "call path".

Every cnode gets as a parameter a region, source file (module), its id and parent cnode (caller).

Parent cnodes have to be defined before their children. Region might be entered from different places in the program, therefore different cnodes might have same region as a parameter.

```
cube_cnode *cnode0, *cnode1, *cnode2;
cnode0 = cube_def_cnode_cs(cube, regn0, mod, 21, NULL);
cnode1 = cube_def_cnode_cs(cube, regn1, mod, 60, cnode0);
cnode2 = cube_def_cnode_cs(cube, regn2, mod, 80, cnode0);
```

CUBE4 supports two kind of parameters of a cnode: numeric and string parameter. Every cnode can carry any number of both of them.

```
cube_cnode_add_numeric_parameter(cnode0, "Phase", 1);
cube_cnode_add_numeric_parameter(cnode0, "Phase", 2);
cube_cnode_add_string_parameter(cnode0, "Iteration", "Initialization");
cube_cnode_add_string_parameter(cnode2, "Etappe", "Finish");
```

Thelast dimension is the `system tree` dimension. Currently CUBE defines the system dimension with the fixed hierarchy: `MACHINE` $\rightarrow$ `NODES` $\rightarrow$ `PROCESSES` $\rightarrow$ `THREADS`

It leads to the fixed sequence of calls in the system dimension definition:

1. First one creates a root for the system dimension : *cube_machine*. Machine has a name and description.
2. Machine consists of *cube_node*s. Every *cube_node* has a name and a *cube_-machine* as a parent.
3. On every *cube_node* run several *cube_process*es (as many cores are available). *cube_process* has a name, MPI rank and *cube_node* as a parent.
4. Every *cube_process* spawns several (one or more) *cube_thread*s (OMP, Pthreads, Java Threads). *cube_thread* has a name, its rank and *cube_process* as a parent.

The cube returns a pointer on *cube_machine*, *cube_node*, *cube_process* or *cube_thread*, which has to be used later to define further level in the system tree or to access the data in the cube.

```
cube_machine* mach  = cube_def_mach(cube, "MSC<<juelich>>", "");
cube_node*    node  = cube_def_node(cube, "Athena<<juropa>>", mach);
cube_process* proc0 = cube_def_proc(cube, "Process 0<<master>>", 0, node);
cube_process* proc1 = cube_def_proc(cube, "Process 1<<worker>>", 1, node);
cube_thread*  thrd0 = cube_def_thrd(cube, "Thread 0<<iterator>>", 0, proc0);
cube_thread*  thrd1 = cube_def_thrd(cube, "Thread 1<<solver>>", 1, proc1);
```

CUBE can carry a set of so called "topologies": mappings `THREAD` → (x, y, z, ...)

Then the GUI is used to visualize every value (*cube_metric*, *cube_cnode*, *cube_thread*) for selected metric and cnode as a 1D, 2D or 3D set of points with the different colors.

First one specifies a number of dimensions (any number is supported), a vector with the sizes in every dimension and its periodicity and creates a structure of type *cube_cartesian*

```
long dimv0[NDIMS] = { 5, 5 };
int periodv0[NDIMS] = { 1, 0 };
cube_cartesian* cart0 = cube_def_cart(cube, NDIMS, dimv0, periodv0);
cube_cart_set_name(cart0, "Application Topology 1");
```

The coordinates are defined like a vector and create a mapping.

```
long coordv[NDIMS] = { 0, 0};
cube_def_coords(cube, cart0, thrd1, coordv);


long dimv1[NDIMS] = { 3, 3 };
int periodv1[NDIMS] = { 1, 0 };
cube_cartesian* cart1 = cube_def_cart(cube, NDIMS, dimv1, periodv1);
cube_cart_set_name(cart1, "MPI Topology 3");

long coordv0[NDIMS] = { 0, 1 };
long coordv1[NDIMS] = { 1, 0 };
cube_def_coords(cube, cart1, thrd0, coordv0);
cube_def_coords(cube, cart1, thrd1, coordv1);
```

The same way one can create any number of topologies. They are shown in the GUI.

```
long dimv2[4]    = { 3, 3, 3, 3 };
int  periodv2[4] = { 1, 0, 0, 0 };
cube_cartesian* cart2 = cube_def_cart(cube, 4, dimv2, periodv2);

long coordv20[4] = { 0, 1, 0, 0 };
long coordv21[4] = { 1, 0, 0 ,0 };
cube_def_coords(cube, cart2, thrd0, coordv20);
cube_def_coords(cube, cart2, thrd1, coordv21);
cube_cart_set_name(cart2,"Second");

long dimv3[14]    = { 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 };
int  periodv3[14] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
cube_cartesian* cart3 = cube_def_cart(cube, 14, dimv3, periodv3);
```

```
    long coordv32[14] = { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2 };
    long coordv33[14] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    cube_def_coords(cube, cart3, thrd0, coordv32);
    cube_def_coords(cube, cart3, thrd1, coordv33);
    cube_cart_set_name(cart3,"Third");
```

Once the dimensions are defined, one fills the cube object with the data. Definition of topologies can be done after filling the cube.

Every data value is specified by three "coordinates": (*cube_metric*, *cube_cnode*, *cube_-thread*)

Note, that *cube_machine*, *cube_node* and *cube_process* are not a "coordinate". They are used only to build up the physical construction of the machine.

Actual writing is done metric-wise and row-wise. First all values of one metric written, then the next metric and so on. No mixing of metrics in this sequence is allowed.

Cube writes data row-wise. It means, for a given cnode, one has to provide an array of values, written in the order of threads in the system dimension.

```
    double sev1[2];

    sev1[0]=123.4;
    sev1[1]=567.9;
    cube_write_sev_row(cube, met0, cnode2, sev1);

    sev1[0]=1123.4;
    sev1[1]=2567.9;
    cube_write_sev_row(cube, met0, cnode1, sev1);

    sev1[0]=-1123.4;
    sev1[1]=3567.9;
    cube_write_sev_row(cube, met0, cnode0, sev1);

    sev1[0]=-123.4;
    sev1[1]=-567.9;
    cube_write_sev_row(cube, met1, cnode0, sev1);

    uint64_t sev2[2];
    sev2[0]=23;
    sev2[1]=26;
    cube_write_sev_row(cube, met2, cnode2, sev2);



    printf("Test file %s complete.\n", cubefile);

    cube_free(cube);
    return 0;
}
```