cube
scalasca

CubeLib 4.8 | Library Usage

Intoduction in Cube tool development guide

November 2022
The Scalasca Development Team
scalasca@fz-juelich.de

JÜLICH
FORSCHUNGSZENTRUM

Attention

The Cube Tool Developer Guide is currently being rewritten and still incomplete. However, it should already contain enough information to get you started and avoid the most common pitfalls.

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);
met1 = cube.def_met( "User time",
                     "Uniq_name2",
                     "INTEGER",
                     "sec", "",
                     "http://www.cs.utk.edu/usr.html",
                     "2nd level",
                     met0);
met2 = cube.def_met( "System time",
                     "Uniq_name3",
                     "INTEGER",
                     "sec",
                     "",
                     "http://www.cs.utk.edu/sys.html",
                    "2nd level",
                    met0);
```

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` → `NODES` → `PROCESSES` → `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.

In order to improve performance of Cube changes in internal data contained were made (since >v4.3.x) and in makes necessary to initialize cube object before feeding it whith the data. This will allocate memory for the data, initialize derived metrics engine and similar.

To initialize cube one calls

```
cube.initialize();
```

After this step one can start to fill the cube object with the data.

It is important to note that it is not possible to change dimensions of cube afterwards.

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;
    }

}
```

www.scalasca.org