cube
scalasca

CubeW 4.10 | Library Usage

Intoduction in Cube tool development guide

November 2026
The Scalasca Development Team
scalasca@fz-juelich.de

JÜLICH
FORSCHUNGSZENTRUM

Attention

The Cube Writer Library 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 **3**
1.1 Quick info about makefile. 3
1.2 Commented source 3

2 Examples of using CUBE c-writer library **5**
2.1 Commented source 5

Bibliography **11**

Copyright © 1998–2026 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.

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 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 seeks while writing.

2.1 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` → `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 : *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;
}
```





www.scalasca.org