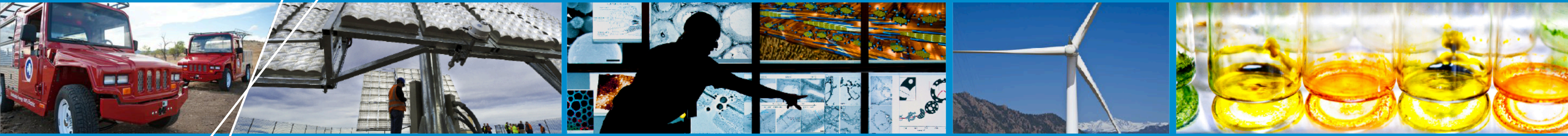
openMDAO DAKOTA plugin



NREL Systems Engineering

Peter Graf et al

Feb 1, 2013

DAKOTA as an openMDAO Driver

Goals:

1) Desire to treat DAKOTA as "just another optimizer"
This usually means user gets to pass data *through* the optimizer to his/her own "callback" that implements the objective function. [but DAKOTA is legitimately complicated]

2) Support hierarchical parallelism (above DAKOTA)—allow providing MPI communicator

3) PYTHON! --- Allow user to call DAKOTA from python (not as a system call). and for DAKOTA to call directly back to python (this is already available in DAKOTA).

4) Expose functionality of DAKOTA as an openMDAO Driver
    Requires passing Driver object *through* DAKOTA
        Callback to objective needs to be orchestrated by Driver's "run_iteration()" method

Ingredients:

$$\begin{tikzcd}\text{DAKOTA ``library mode''} \arrow[r, leftrightarrow, "????"] & \text{openMDAO ``Driver''}\end{tikzcd}$$

step 1: Custom DAKOTA interface

A subclass of DirectApplicInterface [NOTE: DAKOTA 5.2; to be updated for new DAKOTA]

```
class NRELApplicInterface : public DirectApplicInterface
class NRELPythonApplicInterface : public NRELApplicInterface
```

DAKOTA "library mode" that accepts 1) MPI communicator, 2) void * for arbitrary data 3) is aware of our custom interface.

```
int all_but_actual_main_mpi_data(int argc, char* argv[], MPI_Comm comm, void *data);

...
 if (contains(interface.analysis_drivers(),"NREL"))  {
     printf ("replacing interface with NRELApplicInterface\n");
     interface.assign_rep(new NRELApplicInterface(problem_db, data), false);
 ....
```

In DAKOTA input (interface section):

```
analysis_drivers = 'NRELpython'
```

NOTE: Our goals target DAKOTA being called *from* python, so also assumes "interface" is a direct call *to* python:

```
  bp::object * tmp2 = NULL;
  if (pUserData != NULL)
    {
      tmp2 = (bp::object*)pUserData;
      PyDict_SetItem(pDict, PyString_FromString("user_data"),  tmp2->ptr());
    }
```

But this is not strictly necessary.

step 2: python DAKOTA wrapper

---- dakota_python_binding.cpp ----------------------

```
void run_dakota_mpi_data(char *infile, boost::mpi::communicator &_mpi, bp::object data)
{
            int res = all_but_actual_main_mpi_data(2, argv, comm, tmp);
}
BOOST_PYTHON_MODULE(_dakota)
{
            def("run_dakota_mpi_data", run_dakota_mpi_data, "run dakota mpi data");
}
```

---- test_dakface.py ----

```
class AnObj(object):
    """ test object to show we can pass an object through our DAKOTA interface """
    def __init__(self):
        self.mynum = 10

def callback(**kwargs):
    """ callback for rosenbrock test case """
    if ('user_data' in kwargs):
        obj = kwargs['user_data']
        print "magic number is ", obj.mynum
...

if __name__=="__main__":
    import twstr.interfaces.dakota._dakota as _dakota
    _dakota.run_dakota_data("test_dakface.in", AnObj())
```

---- test_dakface.in ----

```
interface,
            direct
              analysis_drivers = 'NRELpython'
              analysis_components = 'test_dakface:callback'             #1,#3
```

(do Demonstration)

step 3: openMDAO DAKOTA Driver

-------- dakota_driver.py ------------

```
class DAKOTADriver(Driver):
          def __init__(self, input_file, *args, **kwargs): ---input_file is the DAKOTA input file
          def execute(self):  --- call DAKOTA from here
          def run_iteration(self):  --- just runs default Driver code!
          def set_values(self,x): -- pass “design parameters” from DAKOTA response function to Driver:
                     self.set_parameters(x)
          def get_responses(self): --- get response back from Driver:
                     return self.eval_objective()

def dakota_callback(**kwargs):
   driver = kwargs['user_data']
   driver.set_values(x)
   driver.run_iteration()
   val = driver.get_responses()
```

----- test_dakota_driver.py -------

```
class AEP_CSM_DAKOTA_Scanner(Assembly):
    """Test assembly that creates a DAKOTA driver and runs a simple aep scan"""

    def __init__(self, nx):
          # Create DAKOTA Driver instance
          driver = DAKOTADriver(self.infile)
          self.add('driver', driver)
...

 dak = AEP_CSM_DAKOTA_Scanner(10)
 set_as_top(dak)
 dak.run()
```

(do Demonstration)

Summary 1: How to use DakotaDriver

1) Instanciate a DakotaDriver instance and pass it the name of the DAKOTA input file (you are responsible for writing):

```
driver = DAKOTADriver(self.infile)
```

2) Create workflow that calculates the objective

```
self.add('aepcomp', aep_csm_assembly())
self.driver.workflow.add('aepcomp')
```

3) Write DAKOTA input file to drive your chosen study: simple helper class: DAKOTAInput. Accepts text strings for DAKOTA input file's standard sections (strategy, method, etc.) as keyword argument

[Interface section uses default: YOU DO NOT NEED TO WRITE THIS

```
interface
            direct
            analysis_drivers = 'NRELpython'
            analysis_components =
'twstr.interfaces.dakota.dakota_driver:dakota_callback’
```

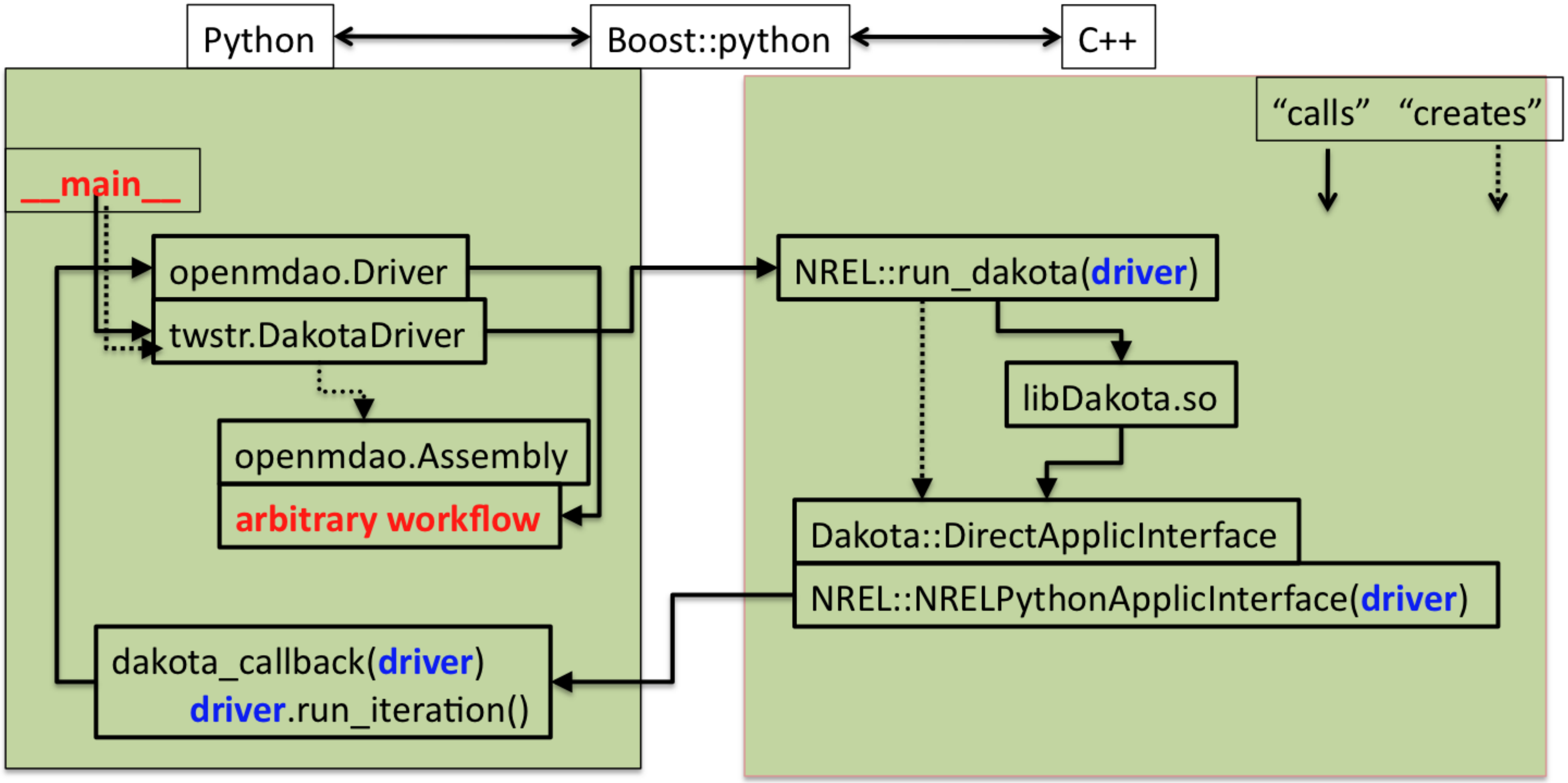
]

4) Run the driver (or assembly that contains it):

```
dak = AEP_CSM_DAKOTA_Scanner(10)
set_as_top(dak)
dak.run()
```

test_dakota_driver.py

Summary 2: DAKOTADriver Architecture



Some instructions to developers

NREL DAKOTA Driver for openMDAO
==============================

Feb, 2013, Peter Graf

This is a _very_ brief description of the materials for this "V-0.0.0.0.1" DakotaDriver.
Please contact me (peter.graf@nrel.gov) for help finishing it off.

C++ source code:
----
dakface.h: header for the C++ materials.

dakface.cpp: This is the library entry point. Following the instructions at http://dakota.sandia.gov/docs/dakota/5.2/html-dev/DakLibrary.html, it detects if the user has chosen the "NRELpython" interface, and installs one if so.

dakota_python_binding.cpp: This is the boost wrapper that exposes the functions in dakface.cpp to python. There are some brief Windows build notes in a comment in this file.

python_interface.cpp: This is where the subclass of DAKOTA's DirectApplicInterface is implemented. I had to copy/paste a lot of code from DAKOTA because the functions I needed were declared "private". However, Brian Adams at DAKOTA has informed me that in the DAKOTA 5.3 release, there is a separate python interface subclass of DirectApplicInterface, and that the relevant functions are now protected (so your subclass can use them). So the new version of this file will be much shorter. The most important change from the regular python interface is that we carry this void * around (in the DAKOTADriver openMDAO object, the void * is the actual driver). Starting on line 198 is where we bundle that up and send it to the python callback.

tdakota_main.cpp: This is main() so that you can run a standalone Dakota that knows about our python interface (ie it calls dakface). I used it mainly for testing.

test_python_binding.cpp: skeletal boost wrapper, for testing.

Mac Makefile

Makefile: Build on the Mac was by a hacked up makefile, here it is.

Boost build files (Windows):

Building on Windows was a royal pain. Your best bet will be to stick to cygwin. But you have to figure out how to get the boost python libraries build correctly, which involves making sure you are picking up the _native_ Windows python libraries, not the ones that are part of cygwin. I _did not_ do it this way, but ended up succeeding by using MinGW. Brian Adams tells me that cygwin is the DAKOTA/Windows build combo of choice for the time being, so I'd suggest figuring out how to make it work.

Jamroot.jam: used to build boost libraries
user-config.jam: tells boost what python to use
boost-build.jam: tells boost where it's build system is.

DAKOTADriver openMDAO Driver:

__init__.py: the usual requirement to be in a package, no code here.
dakota_driver.py: The openMDAO Driver. This should be familiar to you!

misc:
----
make_openmdao_path.py: For running DAKOTA from command line, and having it call back to openMDAO stuff. Needs to know the paths. Not necessary for verion here, which does not use system calls.

Test files for NREL DAKOTADriver openMDAO Driver
===

Feb, 2013, Peter Graf

Brief summary of test materials for DAKOTADriver object:

Basic test, openMDAO NOT required:
----
test_dakface.py: fires up dakota, passes a simple object, verifies that the object comes back through DAKOTA.

test_dakface.in: Dakota input file for this test.

Test of the openMDAO Driver:
----
test_dakota_driver.py: runs a parameter scan of cost of energy, using one of our models. You will maybe want to use a different model for your more generic openMDAO-related tests.

dakota.aepcsm.in: Dakota input file for the test. This file is actuall written _by_ the test, so not needed, but may be a useful reference.