Steering object-oriented computations with Python

T.-Y. B. Yang[1], D. M. Beazley[2], P. F. Dubois[1], and G. Furnish[1]

[1]Lawrence Livermore National Laboratory

[2]Department of Computer Science, University of Utah

1.0 Introduction

Progress has been made in our effort to create a "plug and play" programmable interface for our applications using Numeric Python (Hang *et. al.*, June 1996 Python Conference Proceedings http://www.python.org/workshops/1996-06/papers/). In this progress report we describe our current approach to steering C++ computations with a Python shell. Our experiments have included shadowing C++ objects, combining Tk interfaces with scripting, running Python on parallel processors, and using interface generators. We also include a brief summary of the status of various Python extensions under work at LLNL and LANL (Los Alamos National Laboratory).

2.0 Shadowing C++ Objects and Interface Generation

One of our projects is to develop a physics application in C++ with a Python interface. The application consists of C++ classes grouped into a few modules. The instances of the classes interact with one another to simulate physics processes. The creation and the choreography of these instances (C++ objects) are performed from the Python interpreter. In other words, users can steer the simulations using Python. In order to achieve the steerability, these C++ objects must each have a handle (or shadow) at the Python level. We have implemented an extension type in C named *cplus_shadow* type. For each C++ class of which we wish to create instances from Python, there is a method whose function is to instantiate the class as well as to create a Python shadow for the instance. This creation method is an attribute of the module object to which the class belongs.

The present design makes the shadow object feel and look like the shadowed C++ object as much as possible. If 'x' is a public data member of a class and 'foo' is a shadow object for the class, typing 'foo.x = <new value>' at the Python interpreter changes the value of the data member 'x' of the shadowed C++ object. This is done by the appropriate implementation of the *cplus_shadow* type's *setattr* function, which calls the interface function corresponding to 'x'. If 'x' is of a C++ fundamental type, the interface function uses the proper Python C API (e.g., *PyInt_AsLong* if 'x' is of type 'int') to convert <new value> to the appropriate type. The type conversion also serves as the Run-time type checking. On the other hand, if ‘x’ is declared to be an instance of a C++ class, or a pointer to such an instance, the interface function uses the type information stored in the shadow object to assure that the type of <new value> conforms to the type of ‘x’, i.e. the C++ object shadowed by <new value> is of the same type as ‘x’, or is of a type directly or indirectly inherits from that of ‘x’ through public inheritance. The *getattr* function of the *cplus_shadow* type is also implemented such that the shadow object feels and looks like the shadowed C++ object. Type checking similar to that for public member assignment is also performed for the arguments of C++ public member functions.

Our C++ application also heavily uses templates. Since template classes need to be instantiated during compilation and linking, our current treatment is to write for each template class a separate set of interface functions including the creation function.

For example, if a class template is declared as:

```
template<class T> class Foo {

...

}
```

and we wish to create instances and the corresponding shadow objects for the classes *Foo<A>* and *Foo<B>*, two separate creation functions *hydro_new_Foo_of_A* and *hydro_new_Foo_of_B* are written, where *hydro* is the name of the module. Since the implementations of *hydro_new_Foo_of_A* and *hydro_new_Foo_of_B* involve creation of instances of *Foo<A>* and *Foo<B>,* these two classes are instantiated during compilation and linking.

The following is an example showing how a C++ application can be steered from Python:

```
>>> import Geom, Hydro

>>> mesh = Geom.create_K_Mesh(ncx1=100)

>>> hydro = Hydro.create_Noh_1d("K_Mesh", mesh, \

>>>         Noh_delta=1, gamma=1.6666667 )
```

Two C++ objects and their shadows are created from Python as a result of the above Python statements. The objects are instances of the C++ classes *K_Mesh* and *Noh_1d<K_Mesh>*, respectively. The shadows of these two objects are named ‘mesh’ and ‘hydro’. The file *Hydro.py*, for example, contains the following Python code:

```
import hydro

Error = 'Hydro error'

def create_Noh_1d(template, mesh, **keywords):
```

```
    try:
        ret = getattr(hydro,"new_Noh_1d_of_" + template) (mesh)
    except AttributeError:
        raise Error, \
            "template not instantiated for Noh_1d<"+template+">"
    for kw in keywords.keys():
        try:
            setattr(ret, kw, keywords[kw] )
        except AttributeError:
            raise Error, kw + ' is not an acceptable keyword'
    return ret
```

The module *hydro* is a C++ extension module. One of the attributes of the module object *hydro* is the method *new_Noh_1d_of_K_Mesh* which creates a new instance of the C++ class *Noh_1d<K_Mesh>* as well as the instance's shadow. If another instance of the class template, say *Noh_1d<L_Mesh>*, does not have a corresponding creation method as an attribute of the module object *hydro*. The following statement:

```
>>> hydro = Hydro.create_Noh_1d("L_Mesh", mesh, \
>>>             Noh_delta=1, gamma=1.6666667 )
```

will result in an error with the message 'template not instantiated for Noh_1d<L_Mesh>'. The function *create_Noh_1d* also takes keyword arguments to override default values of the data members of the class *Noh_1d*. If a keyword, say *abc*, which does not correspond to a public data member of *Noh_1d* is given as an argument, an error will be raised with the message 'abc is not an acceptable keyword'.

After the two C++ objects with their shadows named mesh and hydro have been created, the following python script will run the application for 500 steps.

```
t = 0
dt = 0.0000001
for i in range(500):
        if i%10 == 0:
```

```
            print "timestep = " , i, " t = ", t
        hydro.advance (dt)
        t = t + dt
```

We are also learning the interface generating tools SWIG (Simplified Wrapper and Interface Generator) and GRAD (Grammar-based Rapid Application Development). The ways that SWIG and GRAD shadow C++ objects are very similar to each other. For each C++ class, a shadow class, which is a Python class object, is created. A C++ instance of the class is then shadowed by a Python instance object of the Python shadow class. More detailed description of the Python shadow classes is given in the next section. In contrast, the *cplus_shadow* type object in our application is more like the Python instance object with the corresponding class object hidden from the Python interpreter. Although it has not yet been implemented, the *getattr* and *setattr* functions of *cplus_shadow* type can be extended to allow the addition of attributes and methods to the object after its creation, just like the Python instance object. However we shadow the C++ object, the setattr function (or __setattr__ for the Python class) should avoid overriding a method corresponding to a C++ public member function, since such an override results in inconsistent implementations of the class between the points of view of the C++ code and the Python code. This concern is also the reason for our ongoing debate on the merits of having shadow classes which are Python class objects. On the one hand, such shadow classes allows inheritance and addition of new attributes and methods from the Python interpreter. On the other hand, existence of the shadow class objects also allows overriding existing methods for all the instance objects created before or after the override, an undesirable side effect for implementation consistency.

We have also given considerable thought to the interface for pointers to C++ fundamental types. Such pointers are frequently used to point to the addresses for arrays whose shapes are not known during compilation and linking. Although we don't consider this practice a good programming style, we want to be prepared. SWIG and GRAD currently turn a C++ pointer into a built-in Python type object (string for SWIG and long integer for GRAD). Since it takes more than the address of the data to specify a multi-dimensional array, we think this is the best we can do as far as the interface to the compiled code is concerned. However, we plan to write a function which will allow users to create a *PyArray* type object if the users also know the variables specifying the shape of the array (*PyArray* type has been implemented by Jim Hugunin in the Numeric module). The C API *PyArray_FromDimsAndData* in the module can be used to shadow over a data area which is allocated in the compiled code. It will also be useful to implement a function that serves the reverse purpose, i.e., to turn a *PyArray* type object into a tuple containing the Python objects corresponding to the pointers for the data and the shape, respectively, which can be passed as arguments to functions expecting pointer arguments.

3.0 Another Way to Shadow C++ Objects: Python Shadow Class

A Python shadow class is essentially a Python class that is wrapped around an underlying C or C++ class. In doing this, the goal is to produce a Python object that behaves like a normal Python object but also feels a lot like a C/C++ object. For Physics codes, this mapping is especially important since a physicist may be working with various objects in both Python and C/C++ and it is useful for the objects to appear in a similar way in both languages.

As an example, a Physics code might have the following data structures

```
struct Vector {

        double x,y,z;

};

struct Particle {

        Particle();

        ~Particle();

        Vector  r;          // Position

        Vector  v;          // Velocity

        Vector  f;          // Force

        int     type;       // type

};
```

A few very simple Python shadow classes for these structures might look something like the following:

```
import shadow

class Vector:

    def __init__(self, this):

        self.this = this

    def __setattr__(self,name, value):

        if name == "x" :
```

```
            shadow.Vector_x_set(self.this, value)
            return
...
        self.__dict__[name] = value
    def __getattr__(self, name):
        if name == “x” :
            return shadow.Vector_x_get(self.this)
...
        return self.__dict__[name]
class Particle :
    def __init__(self):
        self.this = shadow.new_Particle()
    def __setattr__(self,name,value):
        if name == “r” :
            shadow.Particle_r_set(self.this, value.this)
            return
...
        if name == “type” :
            shadow.Particle_type_set(self.this,value)
            return
        self.__dict__[name] = value
    def __getattr__(self,name):
        if name == “r” :
            return Vector(shadow.Particle_r_get(self.this))
...
```

```
        if name == "type" :
            return shadow.Particle_type_get(self.this)
        return self.__dict__[name]
```

In this case, *shadow* is a module containing a collection of C functions to access various parameters from our data structures. These functions could have been created by SWIG, GRAD, or some other automatic wrapper generator. These C functions have then been encapsulated in a Python class that mirrors the underlying data structures. A pair of getattr and setattr functions provide access to individual members. To construct the objects, one can call a C++ constructor as shown in the Particle class or simply create an object from an already existing pointer as shown for the Vector class.

When used from Python, these shadow classes allow us to do the following:

```
>>> p = Particle()
>>> p.r.x = 0.0
>>> p.r.y = 2.0
>>> p.r.z = 2.5
>>> p.v.x, p.v.y, p.v.z  = (0.0, 0.0, 0.0)
>>> p.f = p.v
>>> p.type = 2
>>> ... etc ...
```

While this is only a simple example, shadow classing can be successfully used in larger systems, e.g., it has been used extensively in Python-controlled large-scale molecular dynamics simulations at Los Alamos National Laboratory.

4.0 Python on Massively Parallel Processing (MPP) Systems

Python uses the C **stdio** library for many operations, including reading scripts from files, importing modules, getting input from the user, and writing byte-compiled versions of modules back to disk. On massively parallel processor (MPP) systems, these operations pose a serious problem. David Beazley have developed an I/O scheme for running Python on MPP systems. His scheme consists of a special header file **pstdio.h** to remap the I/O operations, and I/O wrapper functions implemented using a combination of the C **stdio** library and message passing operations. The header file **pstdio.h** is included into the Python header file prior to the inclusion of the C **stdio.h** header file. This remaps all of the **stdio** operations to the I/O wrapper functions. Two different I/O modes are allowed in this scheme:

- **BROADCAST.** In this mode, processor 0 reads data and broadcast it to all of the other nodes. When writing, output is assumed to come from only one processor. This mode is primarily used for handling interactive I/O using **stdin** and **stdout**.
- **BROADCAST_WRITE.** This mode allows all processors to read data independently, but only one processor can write data. This mode is used for most file operations in Python.

Currently, Beazley has implemented the wrapper under CMMD on the CM-5, the shared memory library on the T3D, and MPI. He is also using Python to control large-scale simulations (molecular dynamics in particular) running on the CM-5 and T3D at Los Alamos National Laboratory.

We have tried this parallel I/O scheme on LLNL T3D and found Beazley's scheme very suitable for our applications. The LLNL T3D has 256 processing elements (PEs), front-ended by a host system, a three processor Cray Y-MP. In our computational model, light-weight Python interpreters, with built-in physics modules, run on the PEs and communicate among themselves using routines provided by the message-passing library. A full-blown Python interpreter, running on the front end host, serves as the user interface. Our users can perform interactive data processing as well as the computation steering on MPP systems utilizing the data processing capability of the front-end Python interpreter and the script parsing capability of the Python enhanced code running on the PEs.

To further elucidate our model, we will use the following scenario as an example. A three-dimensional (3-D) hydrodynamics simulation is being carried out using a domain-decomposition algorithm. Each PE has only detailed information about the domain belonging to the PE but lacks global information about the simulation. The front-end Python interpreter, on the other hand, has a global view of the problem but lacks the most up-to-date detailed information of the simulation. Let's suppose that a user wants to view data on a slice of surface of the 3-D simulation. The front-end host may not have enough memory and disk space to store the information of the 3-D data. Besides, it is a waste of time and space to move all the 3-D data to the front end when the user only wants to view a 2-D slice of the simulation. An efficient way to carry out the user's request is to have the front-end Python interpreter process use its global information to convert the user request into an algorithm described by a Python script (or one Python script for each PE) which is then moved to the PEs. The Python process on each PE parses this script and determines which data on its domain should be transmitted to the front-end host. After receiving the data transmitted by the PEs the front-end Python interpreter can then render the image requested by the user.

One thing worth mentioning is that LLNL T3D is not a time-sharing system, and it is undesirable to leave PEs idle while the user is doing interactive data processing. In the above scenario, conversion of the user request to the Python script is done by the front-end process, which may run on the Cray front-end host or a remote work station. Every so often, while the PEs carry out synchronization among themselves, they also look for Python scripts created by the front-end interpreter. If there is no Python script requesting any service, the PEs will continue their regular computation. Otherwise, the PEs will carry out the service described in the Python script and return to their usual business after finishing the service.

In its preliminary form, the Python script requesting service and the requested data are stored in files and moved between the front-end host and PEs. We plan to explore the possibility of using the Inter-Language Unification (ILU) system as the vehicle for the exchange of information between the front-end host and the PEs.

5.0 Tk Interfaces with Scripting

A Tk interface was built for one of our applications using the Tkinter module. At the click of a button, our application can be initialized or run a few steps. Variables can be plotted versus one another by simply clicking a button. The Tk interface, however, has the disadvantage of being not as programmable as the interpreter. At times, our user may want to do some data processing that was not anticipated when the Tk interface was built. In order to take advantage of both mechanisms, we added an 'Interpreter' button to the Tk interface. At the click of the 'Interpreter' button, the window where the application is evoked becomes the window for the Python interpreter which has the full control of the application as if the Tk interface did not exist. When the user types ^D at the interpreter prompt, the control is returned to the Tk interface. To implement the 'interpreter' button, we added the following method to *physicsmodule.cc* which is one of the C++ extension modules.

```
static PyObject *physics_interpreter(PyObject *self, PyOb-
ject *args)

{

    if (!PyArg_ParseTuple(args, ""))

        return NULL;

    if (PyRun_AnyFile(stdin, "<stdin>") == 0)

        return Py_BuildValue("s", "interpreter ends");

    else

        return Py_BuildValue("s",

             "interpreter ends with an error");

}
```

The Python module where the widgets are implemented contains the following lines of code:

```
from Tkinter import *
```

```
import physics

class Application(Frame):

    def __init__( self, master=None ):

         Frame.__init__( self, master )

         self.createWidgets()

         self.pack()

    def createWidgets( self ):

         self.Interpreter = Button( self, \

            { "text" : "Interpreter", \

            "command" : self.interpreter_back } )

         self.Interpreter.pack()

         ...

   def interpreter_back(self):

         print physics.interpreter()
```

We once considered PTUI (Python/Tkinter User Interface, http://www.lems.brown.edu/~zack/ptui.html) as a tool for the above purpose but could not find an easy way to gain access to the dictionary of the __main__ module. PTUI is a good development tool for applications written in Python. It allows multiple buffers with individual namespaces. For the above purpose, however, we would like the interpreter to have the same namespaces as the Tk interface.

6.0 Summary

We have described our current approaches and future plans for steering C++ applications, running Python on parallel platforms, and combining the Tk interface and the Python interpreter in steering computations. In addition, there has been significant enhancement in the Gist module (to be presented by Zane Motteler). Tk mega widgets has been implemented for a few of our physics applications (to be presented by Geoff Furnish). We have also written a Python interface to SILO, a data storage package used as an interface to a visualization system named MeshTV, developed by another division at LLNL. Python is being used to control large-scale simulations (molecular dynamics in particular) running on the CM-5 and T3D at Los Alamos National Laboratory as well. A few other code development projects at LLNL are either using or considering Python as their steering shells. In summary, the merits of Python have been appreciated by more and more people in the scientific computation community.

Acknowledgments

We are grateful for valuable discussions with Charlie Fly, Robin Friedric, and Greg Boes. The research described here was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.