Dynamic Languages for HPC at LLNL
VEESC Workshop
September 3-4, 2010

T.J. Alumbaugh
Thanks to: Doug Miller, Mike Owen,
Tom Brunner, Patrick Brantley
(LLNL)
Forrest Iandola (UIUC)
Overview

- Examples of applications that use dynamic languages at LLNL
  - Mercury
  - Kull
- Challenges for dynamic languages in HPC
- Performance of Python at scale in HPC environments
- Future directions for Python in HPC
Mercury is a general-purpose parallel Monte Carlo particle transport code written in C++.  

Health physics: a “phantom” human can be modeled (at right) and placed in a scenario. The code can then determine neutron deposition into the skull over the course of a simulation.

At right, the National Ignition Facility target chamber is modeled. This model was used in Mercury to simulate neutron deposition into the surrounding facility walls and evaluate the hazards correspondingly.
A Dynamic Language was added to the existing code to simplify the software development cycle.

The Mercury application embeds Python to make it easier to test and validate the software.

**Goal:** replace the majority of compiled C++ testing with Python scripts for shorter compile times and faster development cycle.

```python
#Call at each cycle of Mercury execution
energyTal = mc.tally.tal["EnergyDeposition"]

if energyTal.getValue(Particle="Neutron", Cell="Skull") > 1e-6:
    print "Neutron energy deposition to the skull reached threshold."
```
Kull is an inertial confinement fusion simulation application

- Massively parallel C++/Python code for inertial confinement fusion
- ~300,000 lines of C++
- Wrapped and exposed to Python via SWIG
- Uses MPI and pypmpi for parallel communication at C++ and Python layers (respectively)
In Kull, the dynamic language is “front and center” and the static language components are compiled, then imported at runtime.

```
>> from kull import *
>> mesh = Mesh(aFileName)
```

The Kull application extends Python to provide a “steerable” simulation code.

Pros:
- flexibility, “it’s just Python”, “like a duck” interface compliance, easy to write tests

Cons:
- High costs (maintenance, compile time, etc.) paid for binding technology
  Ex: ~350K lines of code, 1.7 mil lines of generated wrapper code.
Challenge: automatic binding technologies (e.g. SWIG) incur performance penalties when calling in to other language

Example: SWIG wrapping of basic C++ member function:

```c
static PyObject * _wrap_Ship_getKind(PyObject *self, PyObject *args, PyObject *kwargs) {
    PyObject *resultobj;    Ship *arg1 = (Ship *) 0 ;
    int result;
    PyObject * obj0  = 0 ;
    char *kwnames[] = {        "self", NULL    };

    if(!PyArg_ParseTupleAndKeywords(args,kwargs,(char *)"O:Ship_getKind",kwnames,&obj0))
        goto fail;
    if ((SWIG_ConvertPtr(obj0,(void **) &arg1, SWIGTYPE_p_Ship,SWIG_POINTER_EXCEPTION | 0 )) == -1)
        SWIG_fail;
    result = (int)(arg1)->getKind();
{
    PyObject *module = PyImport_ImportModule("demo");
    if (module != NULL) {
        PyObject *function = PyObject_GetAttrString(module,"enumOutConverter");
        if (function != NULL) {
            PyObject *enumModule = PyImport_ImportModule("demo");
            if (enumModule != NULL) {
                PyObject_CallFunction(function,"Osis", enumModule, "Ship",
```
Challenge: automatic binding technologies (e.g. SWIG) incur performance penalties when calling in to other language.

Example: SWIG wrapping of basic C++ member function:

```c
static PyObject *)__wrap_Ship_getKind(PyObject *self, PyObject *args, PyObject *kwargs) {

    PyObject *resultobj;    Ship *arg1 = (Ship *) 0 ;
    int result;
    PyObject * obj0  = 0 ;
    char *kwnames[] = {        "self", NULL    };

    if(!PyArg_ParseTupleAndKeywords(args,kwargs,(char *)"O:Ship_getKind",kwnames,&obj0))
        goto fail;
    if ((SWIG_ConvertPtr(obj0,(void **) &arg1,
            SWIGTYPE_p_Ship,SWIG_POINTER_EXCEPTION | 0 )) == -1)
        SWIG_fail;
    result = (int)(arg1)->getKind();
{

    PyObject *module = PyImport_ImportModule("demo");
    if (module != NULL) {
        PyObject *function = PyObject_GetAttrString(module, "enumOutConverter");
        if (function != NULL) {
            PyObject *enumModule = PyImport_ImportModule("demo");
            if (enumModule != NULL) {
                resultobj = PyObject_CallFunction(function, "Osis", enumModule,
                    "Ship",
```
Challenge: automatic binding technologies (e.g. SWIG) incur performance penalties when calling in to other language

For performance intensive applications, it is often a good idea to profile the performance of the wrapping technology in various scenarios.
Challenge: interfacing Python with MPI must be done carefully to avoid performance penalties at scale.

Example: pympi collective operations may cause problems at large proc. counts on the BG/P system.
Challenge: interfacing Python with MPI must be done carefully to avoid performance penalties at scale.

Example: pympi collective operations may cause problems at large proc. counts on the BG/P system.
Questions that we care about and don’t know the answers to…

- How can we minimize the performance penalty of binding technologies for existing codes in static languages?
- Which technology has the best interface to MPI?
- Will micro-kernels for computational nodes evolve to meet the needs of dynamic languages? (e.g. CNK for Blue Gene)?
- Will embracing a dynamic language for an application exclude us from running on certain kinds of hardware? (e.g. Roadrunner)?
- Can Python evolve to overcome the limitations of the GIL (global interpreter lock)? What kinds of concurrency solutions will be available in Python? Or some other dynamic language?
- Can we leverage the dynamic nature of Python to adapt our application to use emerging technologies (e.g. pyOpenCL, Theano, etc.)?
- Would it be better to develop in a purely dynamic language and then optimize on the bottlenecks (using Cython, pybindgen, BPL, etc)?
References

- Cython users group on Google Groups for wrapping technology benchmark results:
  http://groups.google.com/group/cython-users/browse_thread/thread/9503bd9468f92447