Scientific Computing with Python
[Advanced Topics]

Eric Jones
eric@enthought.com

Enthought
www.enthought.com

Travis Oliphant
oliphant@ee.byu.edu

Brigham Young University
http://www.ee.byu.edu/
Topics

- Python as Glue
- Wrapping Fortran Code
- Wrapping C/C++
- Parallel Programming
Python as “Glue”
Why Python for glue?

• Python reads almost like “pseudo-code” so it’s easy to pick up old code and understand what you did.
• Python has dynamic typing and dynamic binding --- allows very flexible coding.
• Python is object oriented.
• Python has high-level data structures like lists, dictionaries, strings, and arrays all with useful methods.
• Python has a large module library (“batteries included”) and common extensions covering internet protocols and data, image handling, and scientific analysis.
• Python development is 5-10 times faster than C/C++ and 3-5 times faster than Java
Electromagnetics Example

(1) Parallel simulation
(2) Create plot
(3) Build HTML page
(4) FTP page to Web Server
(5) E-mail users that results are available.
How is Python glue?
Why is Python good glue?

- Python can be embedded into any C or C++ application
  Provides your legacy application with a powerful scripting language instantly.
- Python can interface seamlessly with Java
  - Jython  [www.jython.org](http://www.jython.org)
  - JPE  [jpe.sourceforge.net](http://jpe.sourceforge.net)
- Python can interface with critical C/C++ and Fortran subroutines
  - Rarely will you need to write a main-loop again.
  - Python does not directly call the compiled routines, it uses interfaces (written in C or C++) to do it --- the tools for constructing these interface files are fantastic (sometimes making the process invisible to you).
Tools

- **C/C++ Integration**
  - SWIG [www.swig.org](http://www.swig.org)
  - SIP [www.riverbankcomputing.co.uk/sip/index.php](http://www.riverbankcomputing.co.uk/sip/index.php)
  - Pyrex [nz.cosc.canterbury.ac.nz/~greg/python/Pyrex](http://nz.cosc.canterbury.ac.nz/~greg/python/Pyrex)
  - weave [www.scipy.org/site_content/weave](http://www.scipy.org/site_content/weave)

- **FORTRAN Integration**
  - f2py [cens.ioc.ee/projects/f2py2e/](http://cens.ioc.ee/projects/f2py2e/)
  - PyFort [pyfortran.sourceforge.net](http://pyfortran.sourceforge.net)
• Author: **Pearu Peterson** at Center for Nonlinear Studies Tallinn, Estonia

• Automagically “wraps” Fortran 77/90/95 libraries for use in Python. *Amazing.*

• *f2py* is specifically built to wrap Fortran functions using NumPy arrays.
Simplest f2py Usage

Fortran File

fcopy.f

Python Extension Module

fcopymodule.so

f2py –c fcopy.f –m fcopy

Compile code and build an extension module

Name the extension module fcopy.
Simplest Usage Result

Fortran file `fcopy.f`

```fortran
SUBROUTINE FCOPY(AIN,N,AOUT)

DOUBLE COMPLEX AIN(*)
INTEGER N
DOUBLE COMPLEX AOUT(*)

DO 20 J = 1, N
   AOUT(J) = AIN(J)
20 CONTINUE
END
```

```python
>>> import fcopy
>>> info(fcopy)
This module 'fcopy' is auto-generated with f2py (version:2.37.233-1545).
Functions:
   fcopy(ain,n,aout)
>>> info(fcopy.fcopy)
fcopy - Function signature:
fcopy(ain,n,aout)
Required arguments:
   ain : input rank-1 array('D') with bounds (*)
   n : input int
   aout : input rank-1 array('D') with bounds (*)
```

```python
>>> a = rand(1000) + 1j*rand(1000)
>>> b = zeros((1000,),'D')
>>> fcopy.fcopy(a,1000,b)
```

Looks exactly like the Fortran --- but now in Python!
More Sophisticated

Fortran File
fcopy.f

Interface File
fcopy.pyf
dhand edit

Python Extension Module
fcopymodule.so

f2py fcopy.f -h fcopy.pyf -m fcopy
f2py -c fcopy.pyf fcopy.f
More Sophisticated

Interface file fcopy.pyf
!
-- f90 --
python module fcopy ! in
    interface ! in :fcopy
        subroutine fcopy(ain,n,aout) ! in :fcopy:fcopy.f
            double complex dimension(n), intent(in) :: ain
            integer, intent(hide),depend(ain) :: n=len(ain)
            double complex dimension(n),intent(out) :: aout
        end subroutine fcopy
    end interface
end python module fcopy
!

More Pythonic behavior

```python
>>> a = rand(100,'F')
>>> b = fcopy.fcopy(a)
>>> print b.typecode()
'D'
```

Give f2py some hints as to what these variables are used for and how they may be related in Python.

More Pythonic behavior
ForTRAN File

fcopy.f
hand edit

Python Extension Module

fcopypmodule.so

f2py -c fcopy.f -m fcopy

Compile code and build an extension module

Name the extension module fcopy.
Fortran file fcopy2.f
C
  SUBROUTINE FCOPY(AIN,N,AOUT)
C
  CF2PY INTENT(IN), AIN
  CF2PY INTENT(OUT), AOUT
  CF2PY INTENT(HIDE), DEPEND(A), N=LEN(A)
  DOUBLE COMPLEX AIN(*)
  INTEGER N
  DOUBLE COMPLEX AOUT(*)
  DO 20 J = 1, N
       AOUT(J) = AIN(J)
  20 CONTINUE
END

A few directives can help f2py interpret the source.

```python
>>> a = rand(1000)
>>> import fcopy
>>> info(fcopy.fcopy)
fcopy - Function signature:
aout = fcopy(ain)
Required arguments:
  ain : input rank-1 array('D') with bounds (n)
Return objects:
  aout : rank-1 array('D') with bounds (n)
```

Much more Python like!
Saving the Module C-File

- `f2py -h alib.pyf -m alib *.f`
- `f2py alib.pyf`
- `f2py -c alibmodule.c *.f`
- `f2py -c alibmodule.c -l alib`

Library of Fortran Files

*.*.f

Interface File

`flib.pyf`

hand edited

C-extension Module

`flibmodule.c`

Library

`libflib.a`

Shared extension Module

`flibmodule.so`
Python and Numeric use C conventions for array storage (row major order). Fortran uses column major ordering.

Numeric:

\[ A[0,0], A[0,1], A[0,2], \ldots, A[N-1,N-2], A[N-1,N-1] \]
(last dimension varies the fastest)

Fortran:

\[ A(1,1), A(2,1), A(3,1), \ldots, A(N-1,N), A(N,N) \]
(first dimension varies the fastest)

f2py handles the conversion back and forth between the representations if you mix them in your code. Your code will be faster, however, if you can avoid mixing the representations (impossible if you are calling out to both C and Fortran libraries that are interpreting matrices differently).
How do I distribute this great new extension module?

Recipient must have f2py and scipy_distutils installed (both are simple installs)

Create setup.py file

Distribute *.f files with setup.py file.

Optionally distribute *.pyf file if you’ve spruced up the interface in a separate interface file.

Supported Compilers

g77, Compaq Fortran, VAST/f90 Fortran, Absoft F77/F90, Forte (Sun), SGI, Intel, Itanium, NAG, Lahey, PG
In scipy.stats there is a function written entirely in Python

```python
>>> info(stats.morestats._find_repeats)
_find_repeats(arr)

Find repeats in the array and return a list of the repeats and how many there were.
```

**Goal:** Write an equivalent fortran function and link it in to Python with f2py so it can be distributed with scipy_base (which uses scipy_distutils) and be available for stats.

Python algorithm uses sort and so we will need a fortran function for that, too.
Fortran file futil.f
C     Sorts an array arr(1:N) into
SUBROUTINE DQSORT(N,ARR)
CF2PY INTENT(IN,OUT,COPY), ARR
CF2PY INTENT(HIDE), DEPEND(ARR), N=len(ARR)
  INTEGER N,M,NSTACK
  REAL*8 ARR(N)
  PARAMETER (M=7, NSTACK=100)
  INTEGER I,IR,J,JSTACK, K,L, ISTACK(NSTACK)
  REAL*8 A,TEMP
    ...
END
C     Finds repeated elements of ARR
SUBROUTINE DFREPS(ARR,N,REPLIST,REPNUM,NLIST)
CF2PY INTENT(IN), ARR
CF2PY INTENT(OUT), REPLIST
CF2PY INTENT(OUT), REPNUM
CF2PY INTENT(OUT), NLIST
CF2PY INTENT(HIDE), DEPEND(ARR), N=len(ARR)
  REAL*8 REPLIST(N), ARR(N)
  REAL*8 LASTVAL
  INTEGER REPNUM(N)
  INTEGER HOWMANY, REPEAT, IND, NLIST, NNUM
  ...
END

#Lines added to setup_stats.py
#add futil module
sources = [os.path.join(local_path, 'futil.f')]
name = dot_join(package,'futil')
ext = Extension(name,sources)
config['ext_modules'].append(ext)

#Lines added to morestats.py
# (under stats)
import futil
def find_repeats(arr):
    """Find repeats in arr and return (repeats, repeat_count)""
    v1,v2, n = futil.dfreps(arr)
    return v1[:n],v2[:n]
Try It Out!!

```python
>>> from scipy import *
>>> a = stats.randint(1,30,size=1000)
>>> reps, nums = find_repeats(a)
>>> print reps
[  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.]
>>> print nums
[29 37 29 30 34 39 46 20 30 32 35 42 40 39 35 26 38 33 40 29 34 26 38 45 39 38 29 39 29]
```

New function is 25 times faster than the plain Python version
#!/usr/bin/env python
# File: setup_futil.py

from scipy_distutils.core import Extension

ext = Extension(name = 'futil',
                 sources = ['futil.f'])

if __name__ == '__main__':
    from scipy_distutils.core import setup
    setup(name = 'futil',
           description    = "Utility fortran functions",
           author         = "Travis E. Oliphant",
           author_email   = "oliphant@ee.byu.edu",
           ext_modules    = [ext])

# End of setup_futil.py
Weave
• **weave.blitz()**
  Translation of Numeric array expressions to C/C++ for fast execution

• **weave.inline()**
  Include C/C++ code directly in Python code for on-the-fly execution

• **weave.ext_tools**
  Classes for building C/C++ extension modules in Python
```python
>>> import weave
>>> a=1
>>> weave.inline('std::cout << a << std::endl;',['a'])
sc_f08dc0f70451ecf9a9c9d4d0636de3670.cpp
    Creating library <snip>
    1
>>> weave.inline('std::cout << a << std::endl;',['a'])
    1
>>> a='qwerty'
>>> weave.inline('std::cout << a << std::endl;',['a'])
sc_f08dc0f70451ecf9a9c9d4d0636de3671.cpp
    Creating library <snip>
    qwerty
>>> weave.inline('std::cout << a << std::endl;',['a'])
    qwerty
```
>>> import weave
>>> a = 1
>>> support_code = 'int bob(int val) { return val;}'
>>> weave.inline('return_val = bob(a);', ['a'], support_code=support_code)
Creating library <snip>
1
>>> a = 'string'
>>> weave.inline('return_val = bob(a);', ['a'], support_code = support_code)
sc_19f0a1876e0022290e9104c0cce4f00c1.cpp
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_complied\sc_19f0a1876e0022290e9104c0cce4f00c1.cpp(417) : error C2664: 'bob': cannot convert parameter 1 from 'class Py: :String' to 'int' No user-defined-conversion operator available that can perform this conversion, or the operator cannot be called
Traceback (most recent call last):
  <snip>
weave.build_tools.CompileError: error: command '"C:\Program Files\Microsoft Visual Studio\VC98\BIN\cl.exe"' failed with exit status 2
import string
from weave import ext_tools

def build_ex1():
    ext = ext_tools.ext_module('_ex1')
    # Type declarations– define a sequence and a function
    seq = []
    func = string.upper
    code = ""

        py::tuple args(1);
        py::list result(seq.length());
        for(int i = 0; i < seq.length();i++)
        {
            args[0] = seq[i];
            result[i] = PyEval_CallObject(func,py::tuple(args[0]));
        }
    return_val = result;
    ""

    func = ext_tools.ext_function('my_map',code,['func','seq'])
    ext.add_function(func)
    ext.compile()

try:
    from _ex1 import *
except ImportError:
    build_ex1()
    from _ex1 import *

if __name__ == '__main__':
    print my_map(string.lower,['asdf','ADFS','ADSD'])
Efficiency Issues

PSEUDO C FOR STANDARD NUMERIC EVALUATION

```c
>>> c = a + b + c
tmp1
  tmp2

// c code
// tmp1 = a + b
tmp1 = malloc(len_a * el_sz);
for(i=0; i < len_a; i++)
  tmp1[i] = a[i] + b[i];
// tmp2 = tmp1 + c
tmp2 = malloc(len_c * el_sz);
for(i=0; i < len_c; i++)
  tmp2[i] = tmp1[i] + c[i];
```

FAST, IDIOMATIC C CODE

```c
>>> c = a + b + c

// c code
// 1. loops “fused”
// 2. no memory allocation
for(i=0; i < len_a; i++)
  c[i] = a[i] + b[i] + c[i];
```
MAXWELL’S EQUATIONS: FINITE DIFFERENCE TIME DOMAIN (FDTD), UPDATE OF X COMPONENT OF ELECTRIC FIELD

\[ E_x = \frac{1 - \frac{\sigma_x \Delta t}{2\varepsilon_x}}{1 + \frac{\sigma_x \Delta t}{2\varepsilon_x}} E_x + \frac{\Delta t}{\varepsilon_x} \frac{dH_z}{dy} - \frac{\Delta t}{\varepsilon_x} \frac{dH_y}{dz} \]

PYTHON VERSION OF SAME EQUATION, PRE-CALCULATED CONSTANTS

\[ e_x[:,1:,1:] = c_a_x[:,1:,1:] * e_x[:,1:,1:] \]
\[ \quad + c_b_y_x[:,1:,1:] * (h_z[:,1:,1:] - h_z[:,::-1,1:]) \]
\[ \quad - c_b_z_x[:,1:,1:] * (h_y[:,1:,1:] - h_y[:,1:,::-1]) \]
weave.blitz compiles array expressions to C/C++ code using the Blitz++ library.

WEAVE.BLITZ VERSION OF SAME EQUATION

```python
>>> from scipy import weave
>>> # <instantiate all array variables...>
>>> expr = "ex[:,1:,1:] = ca_x[:,1:,1:] * ex[:,1:,1:]
""" + cb_y_x[:,1:,1:] * (hz[:,1:,1:] - hz[:,,:-1,1:]
""" - cb_z_x[:,1:,1:] * (hy[:,1:,1:] - hy[:,1:,:-1])"

>>> weave.blitz(expr)
< 1. translate expression to blitz++ expression>
< 2. compile with gcc using array variables in local scope>
< 3. load compiled module and execute code>
```
## weave.blitz benchmarks

<table>
<thead>
<tr>
<th>Equation</th>
<th>Numeric (sec)</th>
<th>Inplace (sec)</th>
<th>compiler (sec)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b + c</td>
<td>0.027</td>
<td>0.019</td>
<td>0.024</td>
<td>1.13</td>
</tr>
<tr>
<td>a = b + c + d</td>
<td>0.060</td>
<td>0.037</td>
<td>0.029</td>
<td>2.06</td>
</tr>
<tr>
<td>5 pt. avg filter</td>
<td>0.161</td>
<td>-</td>
<td>0.060</td>
<td>2.68</td>
</tr>
<tr>
<td>FDTD (100x100x100)</td>
<td>0.890</td>
<td>-</td>
<td>0.323</td>
<td>2.75</td>
</tr>
</tbody>
</table>

### Float (4 bytes)

<table>
<thead>
<tr>
<th>Equation</th>
<th>Numeric (sec)</th>
<th>Inplace (sec)</th>
<th>compiler (sec)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b + c</td>
<td>0.128</td>
<td>0.106</td>
<td>0.042</td>
<td>3.05</td>
</tr>
<tr>
<td>a = b + c + d</td>
<td>0.248</td>
<td>0.210</td>
<td>0.054</td>
<td>4.59</td>
</tr>
<tr>
<td>5 pt. avg filter</td>
<td>0.631</td>
<td>-</td>
<td>0.070</td>
<td>9.01</td>
</tr>
<tr>
<td>FDTD (100x100x100)</td>
<td>3.399</td>
<td>-</td>
<td>0.395</td>
<td>8.61</td>
</tr>
</tbody>
</table>

### Double (8 bytes)

- Pentium II, 300 MHz, Python 2.0, Numeric 17.2.0
- Speed-up taken as ratio of scipy.compiler to standard Numeric runs.
Weave case study: An iterative solver for Laplace’s Equation

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \]

PURE PYTHON 2000 SECONDS

```python
for i in range(1, nx-1):
    for j in range(1, ny-1):
        tmp = u[i,j]
        u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                   (u[i, j-1] + u[i, j+1])*dx2)
                   / (2.0*(dx2 + dy2))
        diff = u[i,j] - tmp
        err = err + diff**2
```

Thanks to Prabhu Ramachandran for designing and running this example. His complete write-up is available at:

www.scipy.org/site_content/weave/python_performance.html
weave and Laplace’s equation

**USING NUMERIC**

```
old_u = u.copy()  # needed to compute the error.
u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                 (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
err = sum(dot(old_u - u))
```

**WEAVE.BLITZ**

```
old_u = u.copy()  # needed to compute the error.
expr = """
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                   (u[1:-1, 0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
"""
weave.inline(expr, size_check=0)
err = sum((old_u - u)**2)
```
weave and Laplace’s equation

```python
code = ""
  #line 120 "laplace.py" (This is only useful for debugging)
  double tmp, err, diff;
  err = 0.0;
  for (int i=1; i<nx-1; ++i) {
    for (int j=1; j<ny-1; ++j) {
      tmp = u(i,j);
      u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2 +
               (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv;
      diff = u(i,j) - tmp;
      err += diff*diff;
    }
  }

  return_val = sqrt(err);
  ""

  err = weave.inline(code, ['u','dx2','dy2','dnr_inv','nx','ny'],
                     type_converters = converters.blitz,
                     compiler = 'gcc',
                     extra_compile_args = ['-O3','-malign-double']
)
```
## Laplace Benchmarks

<table>
<thead>
<tr>
<th>Method</th>
<th>Run Time (sec)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Python</td>
<td>1897.0</td>
<td>≈ 0.02</td>
</tr>
<tr>
<td>Numeric</td>
<td>29.0</td>
<td>1.00</td>
</tr>
<tr>
<td>weave.blitz</td>
<td>10.2</td>
<td>2.84</td>
</tr>
<tr>
<td>weave.inline</td>
<td>4.3</td>
<td>6.74</td>
</tr>
<tr>
<td>weave.inline (fast)</td>
<td>2.9</td>
<td>10.00</td>
</tr>
<tr>
<td>Python/Fortran (with f2py)</td>
<td>3.2</td>
<td>9.06</td>
</tr>
<tr>
<td>Pure C++ Program</td>
<td>2.4</td>
<td>12.08</td>
</tr>
</tbody>
</table>

- Debian Linux, Pentium III, 450 MHz, Python 2.1, 192 MB RAM
- Laplace solve for 500x500 grid and 100 iterations
- Speed-up taken as compared to Numeric
SWIG

• Author: David Beazley at Univ. of Chicago

• Automatically “wraps” C/C++ libraries for use in Python. Amazing.

• SWIG uses interface files to describe library functions
  – No need to modify original library code
  – Flexible approach allowing both simple and complex library interfaces

• Well Documented
SWIG Process

Interface File

lib.i

Writing this is your responsibility (kinda)

C Extension File

lib_wrap.c

compile

Library Files

*.h files *.c files

compile

Python Extension Module

libmodule.so
Simple Example

**fact.h**

```c
#ifndef FACT_H
#define FACT_H

int fact(int n);

#endif
```

**example.i**

// Define the modules name
%module example

// Specify code that should
// be included at top of
// wrapper file.
%
{
    #include "fact.h"
%
}

// Define interface. Easy way
// out - Simply include the
// header file and let SWIG
// figure everything out.
%include "fact.h"
Building the Module

LINUX

# Create example_wrap.c file
[ej@bull ej]$ swig –python example.i

# Compile library and example_wrap.c code using
# “position independent code” flag
[ej@bull ej]$ gcc –c -fpic example_wrap.c fact.c
   -I/usr/local/include/python2.1
   -I/usr/local/lib/python2.1/config

# link as a shared library.
[ej@bull ej]$ gcc –shared example_wrap.o fact.o
   -o examplemodule.so

# test it in Python
[ej@bull ej]$ python
   ...
   >>> import example
   >>> example.fact(4)
   24

For notes on how to use SWIG with VC++ on Windows, see
http://www.swig.org/Doc1.1/HTML/Python.html#n2
static PyObject *__wrap_fact(PyObject *self, PyObject *args) {
    PyObject *resultobj;
    int arg0;
    int result;
    /* parse the Python input arguments and extract */

    if(!PyArg_ParseTuple(args,"i:fact",&arg0)) return NULL;
    /* call the actual C function with arg0 as the argument*/
    result = (int)fact(arg0);
    /* Convert returned C value to Python type and return it*/
    resultobj = PyInt_FromLong((long)result);
    return resultobj;
}
SWIG  Example 2

**vect.h**

```c
int* vect(int x, int y, int z);
int sum(int* vector);
```

**vect.c**

```c
#include <malloc.h>
#include "vect.h"
int* vect(int x, int y, int z)
{
    int* res;
    res = malloc(3 * sizeof(int));
    return res;
}
int sum(int* v) {
    return v[0] + v[1] + v[2];
}
```

**example2.i**

Identical to example.i if you replace “fact” with “vect”.

**TEST IN PYTHON**

```python
>>> from example2 import *
>>> a = vect(1, 2, 3)
>>> sum(a)
6  # works fine!

# Let’s take a look at the # integer array a.
>>> a
'_813d880_p_int'
# WHAT THE HECK IS THIS???
```
Complex Objects in SWIG

- SWIG treats all complex objects as pointers.
- These C pointers are mangled into string representations for Python’s consumption.
- This is one of SWIG’s secrets to wrapping virtually any library automatically,
- But… the string representation is pretty primitive and makes it “un-pythonic” to observe/manipulate the contents of the object.
Typemaps

Example:

```c
static PyObject * _wrap_sum(PyObject *self, PyObject *args) {
    if(!PyArg_ParseTuple(args,"O:sum",&arg0))
        return NULL;
    result = (int )sum(arg0);
    return resultobj;
}
```

- Typemaps allow you to insert “type conversion” code into various location within the function wrapper.

- Not for the faint of heart. Quoting David: “You can blow your whole leg off, including your foot!”
The result? Standard C pointers are mapped to NumPy arrays for easy manipulation in Python.

---

**YET ANOTHER EXAMPLE – NOW WITH TYPEMAPS**

```python
>>> import example3
>>> a = example3.vect(1,2,3)
>>> a                 # a should be an array now.
array([1, 2, 3], 'i') # It is!
>>> example3.sum(a)  
6
```

The typemaps used for example3 are included in the handouts.

Another example that wraps a more complicated C function used in the previous VQ benchmarks is also provided. It offers more generic handling 1D and 2D arrays.
Parallel Programming in Python
• Python has threads (sort’a)
• pyMPI(pympi.sf.net/)
• pyre (CalTech)
• PyPAR
  (datamining.anu.edu.au/~ole/pypar/)
• SCIENTIFIC
  (starship.python.net/crew/hinsen)
• COW (www.scipy.org)
Cluster Computing with Python

- cow.py
  - Pure Python Approach
  - Easy to Use
  - Suitable for “embarrassingly” parallel tasks

- pyMPP (Message Passing Interface)
  - Developed by Patrick Miller, Martin Casado et al. at Lawrence Livermore National Laboratories
  - De-facto industry standard for high-performance computing
  - Vendor optimized libraries on “Big Iron”
  - Possible to integrate existing HPFortran and HPC codes such as Scalapack (parallel linear algebra) into Python.
Threads

- Python threads are built on POSIX and Windows threads (hooray!)
- Python threads share a “lock” that prevents threads from invalid sharing
- Threads pass control to another thread
  - every few instructions
  - during blocking I/O (if properly guarded)
  - when threads die
from threading import Thread

- a lower level thread library exists, but this is much easier to use

a thread object can “fork” a new execution context and later be “joined” to another

you provide the thread body either by creating a thread with a function or by subclassing it
• we will work at the prompt!

```python
>>> from threading import *
>>> def f(): print 'hello'
>>> T = Thread(target=f)
>>> T.start()
```
Thread operations

- `currentThread()`
- `T.start()`
- `T.join()`
- `T.getName() / T.setName()`
- `T.isAlive()`
- `T.isDaemon() / T.setDaemon()`
from threading import *
def f(a,b,c): print 'hello',a,b,c
c
T = Thread(target=f, args=(11,22), kw= {'c':c})
T.start()
Subclassing a thread

```python
from threading import *
class myThread(Thread):
    def __init__(self,x,**kw):
        Thread.__init__(self,**kw) #FIRST!
        self.x = x
    def run():
        print self.getName()
        print 'I am running',self.x
T = myThread(100)
T.start()
```

NOTE: Only __init__ and run() are available for overload
CAUTION!

- Threads are really co-routines!
- Only one thread can operate on Python objects at a time
- Internally, threads are switched
- If you write extensions that are intended for threading, use
  - PY_BEGIN_ALLOW_THREADS
  - PY_END_ALLOW_THREADS
COW
Electromagnetic Scattering

**Inputs**
environment, target mesh, and multiple frequencies

**Mem:** KB to Mbytes

**Computation**
- \( N^3 \) CPU
- \( N^2 \) storage

**Time:** a few seconds to days

**Mem:** MB to GBytes

**Outputs**
Radar Cross Section values

**Mem:** KB to MBytes

**Small**

**Large!**

**Small**
Cluster Creation

Master

```python
>>> import scipy.cow
#       [name, port]
>>> machines = [['s0',11500],['s1',11500],['s2',11500]]
>>> cluster = scipy.cow.machine_cluster(machines)
```
Master

```python
>>> cluster = scipy.cow.cluster(machines)
>>> cluster.start()
```

`start()` uses ssh to start an interpreter listening on port 11500 on each remote machine.
# Set a global variable on each of the machines.
>>> cluster['a'] = 1

Python
>>> a = 1

s 0

Python
>>> a = 1

s 1

Python
>>> a = 1

s 2
Master

# Set a global variable on each of the machines.
>>> cluster['a'] = 1
# Retrieve a global variable from each machine.
>>> cluster['a']
(1, 1, 1)
#(s0,s1,s2)
```python
# run a function on each remote machine
>>> import os
>>> cluster.apply(os getpid)
(123, 425, 947)
```
Master

# run a code fragment on each remote machine

```python
>>> cluster.exec_code('import os; pid = os.getpid()',
...                     returns = ('pid',))
(123, 425, 947)
```

```python
>>> import os
>>> os.getpid()
123
```

```python
>>> import os
>>> os.getpid()
425
```

```python
>>> import os
>>> os.getpid()
947
```
```python
# divide task evenly (as possible) between workers
>>> import string
>>> s = ['aa','bb','cc','dd']
>>> cluster.loop_apply(string.upper, loop_var=0, args=(s,) )
('AA','BB','CC','DD')
```
Cluster Method Review

- `apply(function, args=(), keywords=None)`
  - Similar to Python’s built-in `apply` function. Call the given `function` with the specified `args` and `keywords` on all the worker machines. Returns a list of the results received from each worker.

- `exec_code(code, inputs=None, returns=None)`
  - Similar to Python’s built-in `exec` statement. Execute the given `code` on all remote workers as if it were typed at the command line. `inputs` is a dictionary of variables added to the global namespace on the remote workers. `returns` is a list of variable names (as strings) that should be returned after the `code` is executed. If `returns` contains a single variable name, a list of values is returned by `exec_code`. If `returns` is a sequence of variable names, `exec_code` returns a list of tuples.
Cluster Method Review

• `loop_apply(function, loop_var, args=(), keywords=None)`
  - Call `function` with the given `args` and `keywords`. One of the arguments or keywords is actually a sequence of arguments. This sequence is looped over, calling `function` once for each value in the sequence. `loop_var` indicates which variable to loop over. If an integer, `loop_var` indexes the `args` list. If a string, it specifies a keyword variable. The loop sequence is divided as evenly as possible between the worker nodes and executed in parallel.

• `loop_code(code, loop_var, inputs=None, returns=None)`
  - Similar to `exec_code` and `loop_apply`. Here `loop_var` indicates a variable name in the inputs dictionary that should be looped over.
• `ps(sort_by='cpu',**filters)`
  Display all the processes running on the remote machine much like the `ps` Unix command. `sort_by` indicates which field to sort the returned list. Also keywords allow the list to be filtered so that only certain processes are displayed.

• `info()`
  Display information about each worker node including its name, processor count and type, total and free memory, and current work load.
>>> herd.cluster.info()

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>CPU</th>
<th>GHZ</th>
<th>MB TOTAL</th>
<th>MB FREE</th>
<th>LOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>2xP3</td>
<td>0.5</td>
<td>960.0</td>
<td>930.0</td>
<td>0.00</td>
</tr>
<tr>
<td>s1</td>
<td>2xP3</td>
<td>0.5</td>
<td>960.0</td>
<td>41.0</td>
<td>1.00</td>
</tr>
<tr>
<td>s2</td>
<td>2xP3</td>
<td>0.5</td>
<td>960.0</td>
<td>221.0</td>
<td>0.99</td>
</tr>
</tbody>
</table>

>>> herd.cluster.ps(user='ej',cpu='>50')

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>USER</th>
<th>PID</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TOTAL MB</th>
<th>RES MB</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>ej</td>
<td>123</td>
<td>99.9</td>
<td>0.4</td>
<td>3.836</td>
<td>3.836</td>
<td>python...</td>
</tr>
<tr>
<td>s1</td>
<td>ej</td>
<td>425</td>
<td>99.9</td>
<td>0.4</td>
<td>3.832</td>
<td>3.832</td>
<td>python...</td>
</tr>
<tr>
<td>s2</td>
<td>ej</td>
<td>947</td>
<td>99.9</td>
<td>0.4</td>
<td>3.832</td>
<td>3.832</td>
<td>python...</td>
</tr>
</tbody>
</table>
Simple FFT Benchmark

(1) STANDARD SERIAL APPROACH TO 1D FFTs

```python
>>> b = fft(a)  # a is a 2D array: 8192 x 512
```

(2) PARALLEL APPROACH WITH LOOP_APPLY

```python
>>> b = cluster.loop_apply(fft,0,(a,))
```

(3) PARALLEL SCATTER/COMPUTE/GATHER APPROACH

```python
>>> cluster.import_all('FFT')
# divide a row wise amongst workers
>>> cluster.row_split('a',a)
# workers calculate fft of small piece of a and stores as b.
>>> cluster.exec_code('b=fft(a)')
# gather the b values from workers back to master.
>>> b = cluster.row_gather('b')
```
## FFT Benchmark Results

<table>
<thead>
<tr>
<th>Method</th>
<th>CPUs</th>
<th>Run Time (sec)</th>
<th>Speed Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) standard</td>
<td>1</td>
<td>2.97</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(2) loop_apply</td>
<td>2</td>
<td>11.91</td>
<td>0.25</td>
<td>-400%</td>
</tr>
<tr>
<td>(3) scatter/compute/gather</td>
<td>2</td>
<td>13.83</td>
<td>0.21</td>
<td>-500%</td>
</tr>
</tbody>
</table>

### Test Setup:

- The array $a$ is 8192 by 512. **ffts** are applied to each row independently as is the default behavior of the **FFT** module.

- The cluster consists of 16 dual Pentium II 450 MHz machines connected using 100 Mbit ethernet.
## FFT Benchmark Results

<table>
<thead>
<tr>
<th>Method</th>
<th>CPUs</th>
<th>Run Time (sec)</th>
<th>Speed Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) standard</td>
<td>1</td>
<td>2.97</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(2) loop_apply</td>
<td>2</td>
<td>11.91</td>
<td>0.25</td>
<td>-400%</td>
</tr>
<tr>
<td>(3) scatter/compute/gather</td>
<td>2</td>
<td>13.83</td>
<td>0.21</td>
<td>-500%</td>
</tr>
<tr>
<td>(3) compute alone</td>
<td>2</td>
<td>1.49</td>
<td>2.00</td>
<td>100%</td>
</tr>
<tr>
<td>(3) compute alone</td>
<td>4</td>
<td>0.76</td>
<td>3.91</td>
<td>98%</td>
</tr>
<tr>
<td>(3) compute alone</td>
<td>16</td>
<td>0.24</td>
<td>12.38</td>
<td>78%</td>
</tr>
<tr>
<td>(3) compute alone</td>
<td>32</td>
<td>0.17</td>
<td>17.26</td>
<td>54%</td>
</tr>
</tbody>
</table>

**Moral:**
If data can be distributed among the machines once and then manipulated in place, reasonable speed-ups are achieved.
## Electromagnetics

<table>
<thead>
<tr>
<th>EM Scattering Problem</th>
<th>CPUs</th>
<th>Run Time (sec)</th>
<th>Speed Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Buried Sphere</td>
<td>32</td>
<td>8.19</td>
<td>31.40</td>
<td>98.0%</td>
</tr>
<tr>
<td>64 freqs, 195 edges</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Land Mine</td>
<td>32</td>
<td>285.12</td>
<td>31.96</td>
<td>99.9%</td>
</tr>
<tr>
<td>64 freqs, 1152 edges</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
def serial(solver, freqs, angles):
    results = []
    for freq in freqs:
        # single_frequency handles calculation details
        res = single_frequency(solver, freq, angles)
        results.append(res)
    return results

def parallel(solver, freqs, angles, cluster):
    # make sure cluster is running
    cluster.start(force_restart = 0)
    # bundle arguments for loop_apply call
    args = (solver, freqs, angles)
    # looping handled by loop_apply
    results = cluster.loop_apply(single_frequency, 1, args)
    return results
pyMPI
Simple MPI Program

```
# output is asynchronous
% mpirun -np 4 pyMPI
>>> import mpi
>>> print mpi.rank
3
0
2
1

# force synchronization
>>> mpi.synchronizedWrite(mpi.rank, '\n')
0
1
2
3
```
import mpi
import math

if mpi.rank == 0:
    data = [math.sin(x) for x in range(0,10)]
else:
    data = None

common_data = mpi.bcast(data)
• bcast() broadcasts a value from the “root” process (default is 0) to all other processes
• bcast’s arguments include the message to send and optionally the root sender
• the message argument is ignored on all processors except the root
# You can give a little bit to everyone
import mpi
from math import sin, pi
if mpi.rank == 0:
    array = [sin(x*pi/99) for x in range(100)]
else:
    array = None

# give everyone some of the array
local_array = mpi.scatter(array)
mpi.scatter()

- scatter() splits an array, list, or tuple evenly (roughly) across all processors
- the function result is always a list
- an optional argument can change the root from rank 0
- the message argument is ignored on all processors except the root
# Sometimes everyone has a little data to bring
# together
import mpi
import math

local_data = [sin(mpi.rank*x*pi/99) for x in range(100)]
print local_data

root_data = mpi.gather(local_data)
print root_data
mpi.gather() / mpi.allgather()

• gather appends lists or tuples into a master list on the root process
• if you want it on all ranks, use mpi.allgather() instead
• every rank must call the gather()
# You can bring data together in interesting ways

import mpi

x_cubed = mpi.rank**3

sum_x_cubed = mpi.reduce(x_cubed, mpi.SUM)
• The reduce (and allreduce) functions apply an operator across data from all participating processes
• You can use predefined functions
  – mpi.SUM, mpi.MIN, mpi.MAX, etc…
• you can define your own functions too
• you may optionally specify an initial value
3D Visualization with VTK
Visualization with VTK

• Visualization Toolkit from Kitware
  – [www.kitware.com](http://www.kitware.com)

• Large C++ class library
  – Wrappers for Tcl, Python, and Java
  – Extremely powerful, but…
  – Also complex with a steep learning curve
VTK Gallery
VTK Pipeline

Pipeline view from Visualization Studio at http://www.principiamathematica.com
# VTK lives in two modules
from vtk import *

# Create a renderer
renderer = vtkRenderer()

# Create render window and connect the renderer.
render_window = vtkRenderWindow()
render_window.AddRenderer(renderer)
render_window.SetSize(300,300)

# Create Tkinter based interactor and connect render window.
# The interactor handles mouse interaction.
interactor = vtkRenderWindowInteractor()
interactor.SetRenderWindow(render_window)
# Create cone source with 200 facets.
cone = vtkConeSource()
cone.SetResolution(200)

# Create color filter and connect its input
# to the cone's output.
color_filter = vtkElevationFilter()
color_filter.SetInput(cone.GetOutput())
color_filter.SetLowPoint(0,-.5,0)
color_filter.SetHighPoint(0,.5,0)

# map colorized cone data to graphic primitives
cone_mapper = vtkDataSetMapper()
cone_mapper.SetInput(color_filter.GetOutput())
# Create actor to represent our cone and connect it to the mapper
cone_actor = vtkActor()
cone_actor.SetMapper(cone_mapper)

# Assign actor to the renderer.
renderer.AddActor(cone_actor)

# Initialize interactor and start visualizing.
interactor.Initialize()
interactor.Start()
# Convert list of points to VTK structure
verts = vtkPoints()
temperature = vtkFloatArray()
for p in points:
    verts.InsertNextPoint(p[0], p[1], p[2])
temperature.InsertNextValue(p[3])

# Define triangular cells from the vertex
# “ids” (index) and append to polygon list.
polygons = vtkCellArray()
for tri in triangles:
    cell = vtkIdList()
    cell.InsertNextId(tri[0])
    cell.InsertNextId(tri[1])
    cell.InsertNextId(tri[2])
polygons.InsertNextCell(cell)
# Create a mesh from these lists
mesh = vtkPolyData()
mesh.SetPoints(verts)
mesh.SetPolys(polygons)
mesh.GetPointData().SetScalars( ... temperature)

# Create mapper for mesh
mapper = vtkPolyDataMapper()
mapper.SetInput(mesh)

# If range isn’t set, colors are not plotted.
mapper.SetScalarRange( ... temperature.GetRange())

Code for temperature bar not shown.
VTK Demo