Effective Use of Python on Theta

William Scullin
Assistant Computational Scientist
Leadership Computing Facility
Argonne National Laboratory

ALCF Early Science Program Training Series
December 5, 2018

www.anl.gov

“People are doing high performance computing with Python... how do we stop them?”

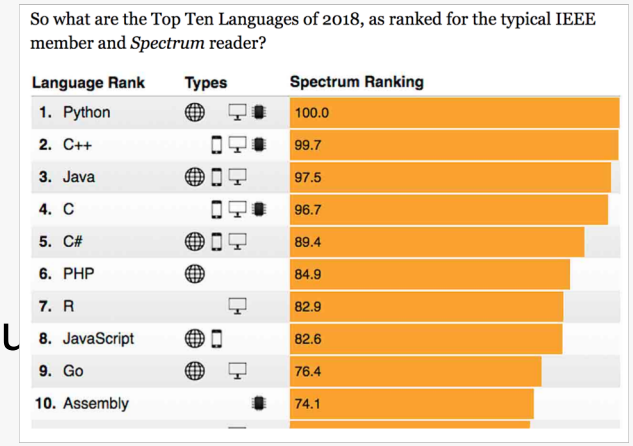
- Senior Performance Engineer

What are we covering?

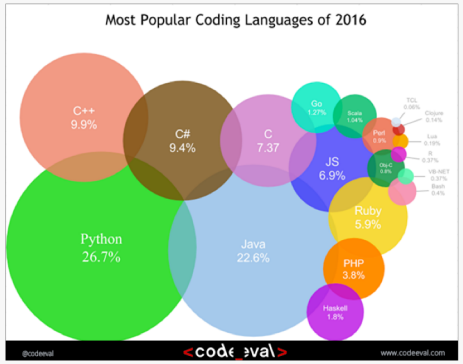
- Why Python?
- Choosing a Python and Environments
- Performance Basics
- NumPy
- Hands-on 1: Setup an environment
- General parallelism
- mpi4py usage
- Job submission with **`aprun`**
- Hands-on 2: Run a parallel job
- Installing and building Python modules
- Hands-on 3: Installing h5py

Why Python?

- If you like a programming paradigm, it's supported
- Most functions map to what you know already
- Easy to combine with other languages
- Easy to keep code readable and maintainable
- Lets you do just about anything without changing langu
- The price is right - no license management
- Code portability
- Fully Open Source
- Very low learning curve
- Commercial support options are available
- Comes with a highly enthusiastic and helpful community


So what are the Top Ten Languages of 2018, as ranked for the typical IEEE member and *Spectrum* reader?

Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C++		99.7
3. Java		97.5
4. C		96.7
5. C#		89.4
6. PHP		84.9
7. R		82.9
8. JavaScript		82.6
9. Go		76.4
10. Assembly		74.1



Why Not Python?

- Performance is often a secondary concern for developers and distributions
  - Most developers aren't in HPC environments
  - Most developers aren't in science environments
- Many tools were designed to work best in generic environments
- Language maintainers favor consistency over compatibility
- Backwards compatibility is seldom guaranteed
- Low learning curve
- It's easy to develop a code base that works, but won't scale

Python 2 or 3? In general, use Python 3.

- **Python 3 is the future** – and the future is here
- All **major libraries now work** under Python 3.5+
- Almost **all popular tools work** with Python 3.5+
- Python 3's loader and more of the interpreter's internals are written in Python which does make it slower in distributed environments
- Python 2 development has effectively stopped

Python at ALCF

- Every system we run is a cross-compile environment except Cooley
  - pip/distutils/setuptools/anaconda don't play well with cross-compiling
- Blue Gene/Q Python is manually maintained:
  - Instructions on use are available in: `/soft/cobalt/examples/python`
  - Modules built on request, but BG/Q is end-of-life
- X86_64 offers us a lot more options:
  - Miniconda
  - Intel Python - managed and used via Conda
  - ALCF Python managed via Spack and loadable via modules
  - Bring your own Python
- We prefer users to install their own environments
- Users will need to set up their environment to use the Cray MPICH compatibility ABI and strictly build with the Intel MPI wrappers: http://docs.cray.com/books/S-2544-704/S-2544-704.pdf

Python at ALCF

- Conda-based options:
  - Theta Miniconda

```
module avail 2>&1 | grep miniconda
miniconda-2.7/conda-4.4.10
miniconda-2.7/conda-4.4.10-h5py-parallel
miniconda-2.7/conda-4.4.10-login
miniconda-2.7/conda-4.5.4
miniconda-2.7/conda-4.5.4-login
miniconda-3.6/conda-4.4.10
miniconda-3.6/conda-4.4.10-login
miniconda-3.6/conda-4.5.4
miniconda-3.6/conda-4.5.4-login
```

  - Intel Python - managed and used via Conda
  - Anaconda

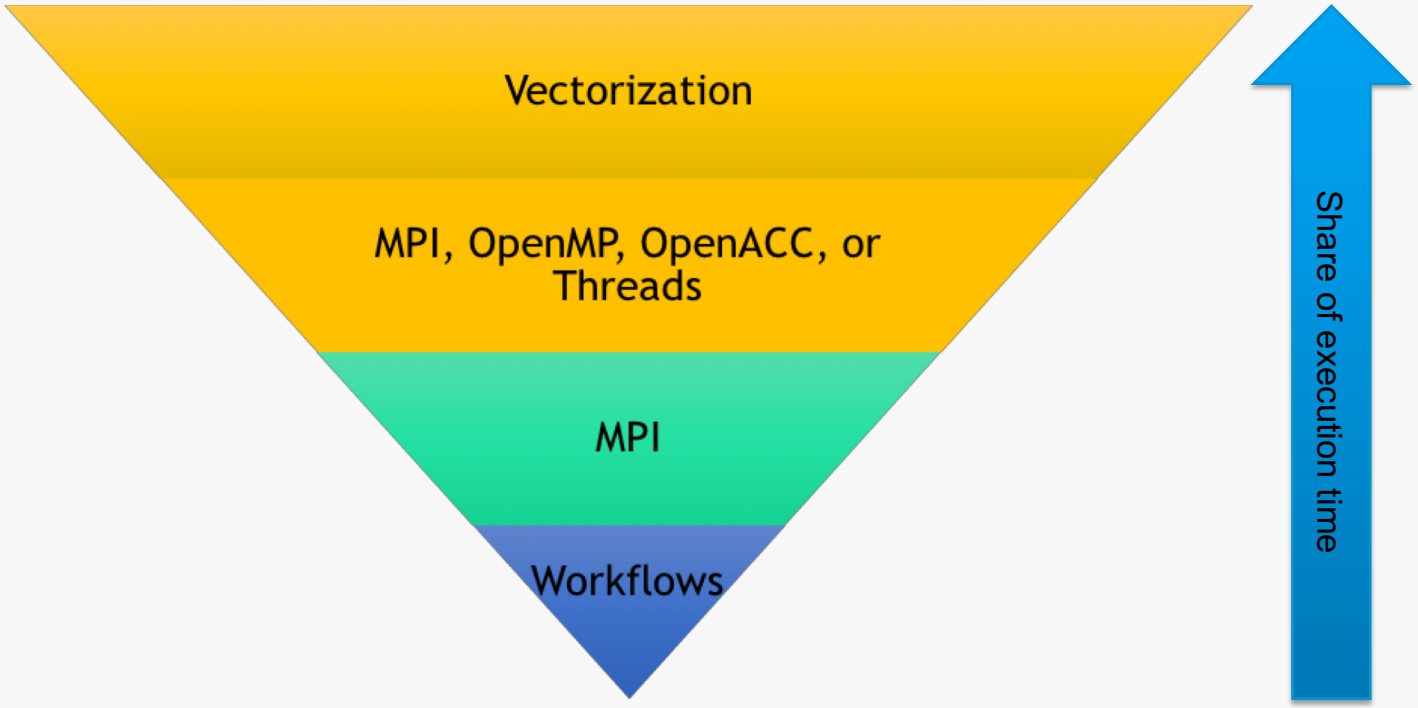
Python at ALCF

- Built-from-source Python
  - ALCF Python managed via Spack and loadable via modules:
    `module load alcfpython/2.7.14-20180131`
    - A module that loads modules for NumPy, SciPy, MKL, h5py, mpi4py...
    - Built via Spack to emphasize performance, reproducibility, and Cray compatibility
    - Use of virtualenv is recommended - **do not mix conda and virtualenv**!
    - We'll build any package with a Spack spec on request

Python at ALCF: How to choose?

- We're out to enable you to work in a way that's comfortable for you
  - If you're using Anaconda, use Anaconda
    - we recommend cloning local miniconda and then the Intel channels
    - It's easy to clobber a working environment
    - Be cognizant the Cray MPI requires some manipulation of your envs
  - If you're new, **`VirtualEnv`** is standard outside of the data science community
    - Not as reproducible or sharable in science contexts
    - Universally supported
    - Easy to generate a non-performant build

Where do We want to spend our time?



How does CPython work?

```
Python 2.7.13 (default, Apr 23 2017, 16:50:50)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def area_circle(r):
...     pi=3.14159
...     area=pi*r**2
...     return area
...
>>> import dis
>>> dis.dis(area_circle.func_code)
  2           0 LOAD_CONST               1 (3.14159)
              3 STORE_FAST               1 (pi)

  3           6 LOAD_FAST                1 (pi)
              9 LOAD_FAST                0 (r)
             12 LOAD_CONST               2 (2)
             15 BINARY_POWER
             16 BINARY_MULTIPLY
             17 STORE_FAST               2 (area)

  4          20 LOAD_FAST                2 (area)
             23 RETURN_VALUE
>>>
```

How does CPython work?

```
>>> def area_circles(R):
...     A=[]
...     for r in R:
...         A.append(area_circle(r))
...     return A
...
>>> dis.dis(area_circles.func_code)
  2           0 BUILD_LIST               0
              3 STORE_FAST               1 (A)

  3           6 SETUP_LOOP              33 (to 42)
              9 LOAD_FAST                0 (R)
             12 GET_ITER
        >>   13 FOR_ITER                25 (to 41)
             16 STORE_FAST               2 (r)

  4          19 LOAD_FAST                1 (A)
             22 LOAD_ATTR                0 (append)
             25 LOAD_GLOBAL              1 (area_circle)
             28 LOAD_FAST                2 (r)
             31 CALL_FUNCTION            1
             34 CALL_FUNCTION            1
             37 POP_TOP
             38 JUMP_ABSOLUTE           13
        >>   41 POP_BLOCK

  5     >>   42 LOAD_FAST                1 (A)
             45 RETURN_VALUE
>>>
```

How does CPython work?

```
>>> def area_circles_lc(R):
...     return [area_circle(r) for r in R]
...
>>> dis.dis(area_circles_lc.func_code)
  2           0 BUILD_LIST               0
              3 LOAD_FAST                0 (R)
              6 GET_ITER
        >>    7 FOR_ITER                18 (to 28)
             10 STORE_FAST               1 (r)
             13 LOAD_GLOBAL              0 (area_circle)
             16 LOAD_FAST                1 (r)
             19 CALL_FUNCTION            1
             22 LIST_APPEND              2
             25 JUMP_ABSOLUTE            7
        >>   28 RETURN_VALUE
>>>
```

Threads and CPython: A Word on the GIL

To keep memory coherent, Python only allows a single thread to run in the interpreter's memory space at once. This is enforced by the Global Interpreter Lock, or GIL.

The GIL isn’t all bad. It:

- Is mostly sidestepped for I/O (files and sockets)
- Makes writing modules in C much easier
- Makes maintaining the interpreter much easier
- Makes for any easy topic of conversation
- Encourages the development of other paradigms for parallelism
- Is almost **entirely irrelevant in the HPC space** as it neither impacts MPI or threading within compiled modules

For the gory details, see David Beazley's talk on the GIL: https://www.youtube.com/watch?v=fwzPF2JLoeU

Takeaways on CPython

- CPython is a **Read–Eval–Print Loop** (**REPL**) environment.
- There is no look-ahead to enable optimizations.
- There is no automatic parallelism.
- Everything is evaluated piece-wise and sequentially.
- CPython was written for safety and ease of maintenance, not performance:
  - Russell Power and Alex Rubinsteyn wrote in their paper "How fast can we make interpreted Python?":

    "In the general absence of type information, almost every instruction must be treated as INVOKE_ARBITRARY_METHOD."

- While you can improve pure Python performance through language features running in CPython, it won't deliver the efficiency of compiled code.

Numpy and Scipy

NumPy - your first stop for performance improvement. It provides:

- N-dimensional homogeneous arrays (ndarray)
- Universal functions (ufunc)
- built-in linear algebra, FFT, PRNGs
- Tools for integrating with C/C++/Fortran
- Heavy lifting done by optimized C/Fortran libraries such as Intel's MKL or IBM's ESSL

SciPy extends NumPy with common scientific computing tools

- optimization
- additional linear algebra
- integration
- interpolation
- FFT
- signal and image processing
- ODE solvers

Problems arise when NumPy isn't well built… and its configuration is used for most other scientific modules

Checking your NumPy Configuration:

Check your configuration for the use of optimized libraries:

```
>>> import numpy as np
>>> np.__config__.show()
```

NumPy's distutils can give insight into compilers and options used:

```
>>> import numpy
>>> import numpy.distutils
>>> np_config_vars = numpy.distutils.unixccompiler.sysconfig.get_config_vars()
>>> # np_config_vars is a dict with configuration values
>>> import pprint
>>> # pprint is a pretty printer and not required, just recommended
>>> pprint.pprint(np_config_vars)
{'AC_APPLE_UNIVERSAL_BUILD': 0,
 'AIX_GENUINE_CPLUSPLUS': 0,
 'AR': 'ar',
 'ARCH': 'x86_64',
 'ARFLAGS': 'rc',
...
```

NumPy and SciPy

Optimized and built with MKL via Spack

```
[wscullin@thetalogin6 ~]$ python
Python 2.7.13 (default, May  2 2017, 20:30:06)
[GCC Intel(R) C++ gcc 4.9.4 mode] on linux2
Type "help", "copyright", "credits" or "license" for more information.
readline: /etc/inputrc: line 19: term: unknown variable name
>>> import numpy as np
>>> np.__config__.show()
lapack_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl/lib/intel64']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl', '/projects/
    datascience/soft/builds/spack/packages/opt/linux/mkl/include', '/projects/datascience/soft/
    builds/spack/packages/opt/linux/mkl/lib']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl/lib/intel64']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl', '/projects/
    datascience/soft/builds/spack/packages/opt/linux/mkl/include', '/projects/datascience/soft/
    builds/spack/packages/opt/linux/mkl/lib']
lapack_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl/lib/intel64']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl', '/projects/
    datascience/soft/builds/spack/packages/opt/linux/mkl/include', '/projects/datascience/soft/
    builds/spack/packages/opt/linux/mkl/lib']
blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl/lib/intel64']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl', '/projects/
    datascience/soft/builds/spack/packages/opt/linux/mkl/include', '/projects/datascience/soft/
    builds/spack/packages/opt/linux/mkl/lib']
```

Installed via pip

```
Python 2.7.5 (default, Nov  6 2016, 00:28:07)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> np.__config__.show()
lapack_info:
  NOT AVAILABLE
lapack_opt_info:
  NOT AVAILABLE
openblas_lapack_info:
  NOT AVAILABLE
blas_info:
  NOT AVAILABLE
atlas_3_10_blas_threads_info:
  NOT AVAILABLE
atlas_threads_info:
  NOT AVAILABLE
blas_src_info:
  NOT AVAILABLE
atlas_3_10_threads_info:
  NOT AVAILABLE
atlas_blas_info:
  NOT AVAILABLE
atlas_3_10_blas_info:
  NOT AVAILABLE
lapack_src_info:
  NOT AVAILABLE
atlas_blas_threads_info:
  NOT AVAILABLE
openblas_info:
  NOT AVAILABLE
blas_mkl_info:
  NOT AVAILABLE
blas_opt_info:
  NOT AVAILABLE
blis_info:
  NOT AVAILABLE
atlas_info:
  NOT AVAILABLE
atlas_3_10_info:
  NOT AVAILABLE
lapack_mkl_info:
  NOT AVAILABLE
>>>
```

The test on a KNL system:

```
>>> import timeit
>>> sum([timeit.timeit('import numpy as np; np.random.random((100,100))*np.random.random((100))') for i in range(100)])/100.0
```

119.68859601020813s

499.9269280433655s

NumPy Data Types

NumPy covers all the same numeric data types available in C/C++ and Fortran as variants of int, float, and complex:

- all available signed and unsigned as applicable
- available in standard lengths
- floats are double precision by default
- generally available with names similar to C or Fortran
  -- ie: long double is `longdouble`
- generally compatible with Python data types
- follow endianness of the platform – conversion routines are offered
- `longdouble` follows the compiler / platform's definition of long double

NumPy also offers the ability to create structured datatypes.

If it can be done in C/C++/Fortran, it can be done in NumPy.

Creating NumPy Arrays

```
# Initialize with Python lists: array with 2 rows, 4 cols
>>> import numpy as np
>>> np.array([[1,2,3,4],[8,7,6,5]])
array([[1, 2, 3, 4],
       [8, 7, 6, 5]])

# Make an array of n (10) evenly spaced numbers over an interval
# inclusive of start (1) and stop (100)
>>> np.linspace(1,100,10)
array([   1.,   12.,   23.,   34.,   45.,   56.,   67.,   78.,   89.,  100.])

# Create an array and pre-populate with zeros with 2 rows, 5 cols
>>> np.zeros((2,5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

Slicing NumPy Arrays (Part 1)

```
>>> a = np.array([[1,2,3,4],[9,8,7,6],[1,6,5,4]])
>>> a
array([[1, 2, 3, 4],
       [9, 8, 7, 6],
       [1, 6, 5, 4]])

>>> arow = a[0,:] # get slice referencing row zero
>>> arow
array([1, 2, 3, 4])

>>> cols = a[:,[0,2]] # get slice referencing columns 0 and 2
>>> cols
array([[1, 3],
       [9, 7],
       [1, 5]])
```

Slicing NumPy Arrays (Part 2)

```
# NOTE: arow & cols are NOT copies, they point to the original data
>>> arow
array([1, 2, 3, 4])
>>> arow[:] = 0
>>> arow
array([0, 0, 0, 0])

>>> a
array([[0, 0, 0, 0],
       [9, 8, 7, 6],
       [1, 6, 5, 4]])

# Explicitly copy data
>>> copyrow = arow.copy()
```

Creating NumPy Arrays

```
# Make a 2d n x n (4 x 4) array of 1s
>>> b = np.ones((4,4))
>>> b
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])

>>> b.ndim
2

>>> b.dtype
dtype('float64')

>>> b.shape
(4, 4)
```

Creating NumPy Arrays

```
# Make a 2d n x n (4 x 4) identity array
>>> c = np.eye(4)
>>> c
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])

# Make a 2d n x n (4 x 4) from a function
>>> def f(x,y): return (1/(x+1))*y
...
>>> d = np.fromfunction(f,(4,4))
>>> d
array([[0.        , 1.        , 2.        , 3.        ],
       [0.        , 0.5       , 1.        , 1.5       ],
       [0.        , 0.33333333, 0.66666667, 1.        ],
       [0.        , 0.25      , 0.5       , 0.75      ]])
```

Broadcasting with universal functions (ufuncs)

Applies operations to many elements with a single call – with compiled code

```
>>> a = np.array(([1,2,3,4],[8,7,6,5]))
>>> a
array([[1, 2, 3, 4],
       [8, 7, 6, 5]])
```

Rule 1: Dimensions of one may be prepended to either array to match the array with the greatest number of dimensions

```
>>> a + 1 # add 1 to each element in array
array([[2, 3, 4, 5],
       [9, 8, 7, 6]])
```

Rule 2: Arrays may be repeated along dimensions of length 1 to match the size of a larger array

```
>>> a + np.array(([1],[10])) # add 1 to 1st row, 10 to 2nd row
array([[ 2, 3, 4, 5],
       [18, 17, 16, 15]])
>>> a**([2],[3]) # raise 1st row to power 2, 2nd to 3
array([[ 1,   4,   9, 16],
       [512, 343, 216, 125]])
```

Broadcasting with universal functions (ufuncs)

```
# Beware of matrix versus array syntax
>>> c*d
array([[0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.5       , 0.        , 0.        ],
       [0.        , 0.        , 0.66666667, 0.        ],
       [0.        , 0.        , 0.        , 0.75      ]])
>>> c@d
array([[0.        , 1.        , 2.        , 3.        ],
       [0.        , 0.5       , 1.        , 1.5       ],
       [0.        , 0.33333333, 0.66666667, 1.        ],
       [0.        , 0.25      , 0.5       , 0.75      ]])
>>> c.dot(d) # Equivalent to @ operator for 2d arrays
array([[0.        , 1.        , 2.        , 3.        ],
       [0.        , 0.5       , 1.        , 1.5       ],
       [0.        , 0.33333333, 0.66666667, 1.        ],
       [0.        , 0.25      , 0.5       , 0.75      ]])
>>> np.matmul(c,d) # Different rules from np.dot
array([[0.        , 1.        , 2.        , 3.        ],
       [0.        , 0.5       , 1.        , 1.5       ],
       [0.        , 0.33333333, 0.66666667, 1.        ],
       [0.        , 0.25      , 0.5       , 0.75      ]])
```

Using NumPy appropriately pays off

```
>>> import timeit
>>> import numpy as np
>>> A = np.linspace(-10,10,100).reshape(10,10)
>>> B = np.linspace(-1.0,1.0,100).reshape(10,10)
>>>
>>> def mat_mult(A,B):
...     """ We're assuming regular 2D NumPy matrixes with dimensions such that
...         A.shape[1] == B.shape[0]"""
...     assert A.shape[1] == B.shape[0], "A[1].shape != B[0].shape"
...     C=np.zeros((A.shape[0],B.shape[1]))
...     for i in range(A.shape[0]):
...         for j in range(A.shape[1]):
...             for k in range(B.shape[1]):
...                 C[i,j] += A[i,k]*B[k,j]
...     return C
...
...
>>> if __name__ == '__main__':
...     setup_str = "from __main__ import  A,B,mat_mult; import numpy as np"
...     cnt = 100000
...     manual_time = timeit.timeit("mat_mult(A,B)", number=cnt, setup=setup_str)
...     numpy_time  = timeit.timeit("np.matmul(A,B)", number=cnt, setup=setup_str)
...     print("Manual Matmul x%d: %24.6fs" %(cnt, manual_time))
...     print("NumPy  Matmul x%d: %24.6fs" %(cnt, numpy_time))
...
Manual Matmul x100000:                409.429088s
NumPy  Matmul x100000:                  1.660264s
```

When NumPy isn't enough

- Building blocks like NumPy and SciPy are already built with great vectorizations and thread support via the libraries they link with:
  BLAS/LAPACK, MKL, FFTW
- Don't re-implement solvers in pure Python or even NumPy - many of your favorite libraries and packages already have Python bindings:
  - PyTrilinos
  - petsc4py
  - Elemental
  - SLEPc
- Where bindings for a library aren't available, it's often easy to generate them

Hands-on 1

Notes:

- We have a Theta reservation for use in this training, the queue is `training`
- Examples assume exactly 8 nodes unless specified
- Ask questions if you get confused or something breaks

We'll create a VirtualEnv environment, a Conda env, list packages, and install a mpi4py.

Please log into Theta now.

Hands on – follow live - VirtualEnv:

First, create a directory for this training, I used /home/wscullin/esptraining

```
wscullin@thetalogin5:~> mkdir esptraining
wscullin@thetalogin5:~> cd esptraining
```

Next, check for a Python in your path - it's likely the system Python – we don't recommend using this

```
wscullin@thetalogin5: ~/esptraining> which python
/usr/bin/python
```

Load the alcfpython module

```
wscullin@thetalogin5:~/esptraining> module avail 2>&1 | grep alcfpython
alcfpython/2.7.14-20180131
module load alcfpython/2.7.14-20180131
wscullin@thetalogin5:~/esptraining> which python
/lus/theta-fs0/software/packaging/spack/builds/cray-CNL-mic_knl/intel-18.0.0.128.4.9.4.6.0.4.7.7.0/python-
2.7.14-raulaayvkwjengfs4yk53wmp4nu7y2ls/bin/python
```

Create a VirtualEnv that builds off the central install

```
wscullin@thetalogin5:~/esptraining> virtualenv --system-site-packages training_env
New python executable in /gpfs/mira-home/wscullin/esptraining/training_env/bin/python2.7
Also creating executable in /gpfs/mira-home/wscullin/esptraining/training_env/bin/python
Installing setuptools, pip, wheel...done.
(training_env) wscullin@thetalogin5:~/esptraining> which python
/gpfs/mira-home/wscullin/esptraining/training_env/bin/python
```

Hands on – follow live - VirtualEnv:

List packages in the training_env VirtualEnv

```
(training_env) wscullin@thetalogin5:~/esptraining> pip list
Package                           Version
--------------------------------- -----------
alabaster                         0.7.10
...
```

Check the numpy build location

```
(training_env) wscullin@thetalogin5:~/esptraining> python
Python 2.7.14 (default, Feb  3 2018, 00:03:51)
[GCC Intel(R) C++ gcc 4.9.4 mode] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.__file__
'/lus/theta-fs0/software/packaging/spack/builds/cray-CNL-mic_knl/intel-18.0.0.128.4.9.4.6.0.4.7.7.0/py-numpy-
1.13.3-7eg5w35lsvyiqpiqb6yj7ozaib25y2ip/lib/python2.7/site-packages/numpy/__init__.pyc'
>>> exit()
```

Close things down

```
(training_env) wscullin@thetalogin5:~/esptraining> deactivate
wscullin@thetalogin5:~/esptraining>
wscullin@thetalogin5:~/esptraining> module unload alcfpython
wscullin@thetalogin5:~/esptraining> which python
/usr/bin/python
```

Note on using VirtualEnv

Using your packages with an external interpreter:

- Install your own packages in your virtualenv
- Use them with external python within your python scripts
- Mix-and-match with center-provided packages

Activate automatically in scripts with:

```
#!/usr/bin/env python2.7
activate_this = '/path/to/env/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

N.B.: Packages installed in your virtualenv will supercede versions installed at the site level.

Hands on – follow live - miniconda:

Find the miniconda modules

```
wscullin@thetalogin5:~/esptraining> module avail miniconda

--------------------------- /soft/environment/modules/modulefiles ---------------------------
miniconda-2.7/conda-4.4.10                miniconda-3.6/conda-4.4.10
miniconda-2.7/conda-4.4.10-h5py-parallel miniconda-3.6/conda-4.4.10-login
miniconda-2.7/conda-4.4.10-login          miniconda-3.6/conda-4.5.4
miniconda-2.7/conda-4.5.4                 miniconda-3.6/conda-4.5.4-login
miniconda-2.7/conda-4.5.4-login
```

Load the login version of the module and confirm the python in use

```
wscullin@thetalogin5:~/esptraining> module load miniconda-2.7/conda-4.5.4
wscullin@thetalogin5:~/ esptraining > which python
/soft/datascience/conda/miniconda2/4.5.4/bin/python
```

List packages

```
wscullin@thetalogin5:~/esptraining> conda list
# packages in environment at /soft/datascience/conda/miniconda2/4.5.4:
#
# Name                    Version                   Build  Channel
absl-py                   0.3.0                    py27_0
. . .
```

Hands on – follow live:

Create a Conda environment

```
wscullin@thetalogin5:~/esptraining> conda create -p ./training_conda_env --clone $MINICONDA_INSTALL_PATH
Source:      /soft/datascience/conda/miniconda2/4.5.4
Destination: /gpfs/mira-home/wscullin/esptraining/training_conda_env
The following packages cannot be cloned out of the root environment:
 - conda-4.5.9-py27_0
 - conda-env-2.6.0-h36134e3_1
Packages: 94
Files: 6468
[10 minutes worth of installation messages]
```

Activate the environment

```
wscullin@thetalogin5:~/esptraining> source activate training_conda_env
(/gpfs/mira-home/wscullin/esptraining/training_conda_env) wscullin@thetalogin5:~/esptraining>
```

Verify you're using your python environment

```
(/gpfs/mira-home/wscullin/esptraining/training_conda_env) wscullin@thetalogin5:~/esptraining> which python
/gpfs/mira-home/wscullin/esptraining/training_conda_env/bin/python
```

Close things down

```
(/gpfs/mira-home/wscullin/esptraining/training_conda_env) wscullin@thetalogin5:~/esptraining> source deactivate
wscullin@thetalogin5:~/esptraining>
```

Notes on using Conda

- See https://www.alcf.anl.gov/user-guides/conda for more details
- If you notice that packages installed in your env aren't being chosen over defaults from `$MINICONDA_INSTALL_PATH` you may need to do something like:

```
PV=$(python –c 'import sys; print("%d.%d" %sys.version_info[0:2])')
export PYTHONPATH=${CONDA_PREFIX}/lib/python${PV}/site–packages:${PYTHONPATH}
```

in your shell and scripts

Parallelism

Parallel and Distributed Programming Options

threading

- useful for certain concurrency issues, not really usable for parallel computing due to the GIL

subprocess

- relatively low level control for spawning and managing processes, think popen

multiprocessing - multiple Python instances (processes)

- basic multiple process parallelism through forked interpreters
- **Does not mix well with OpenMP, MPI, or shared memory tools**

MPI

- mpi4py exposes your full local MPI API within Python
- as scalable as your local MPI

GPU (OpenCL & CUDA)

- PyOpenCL and PyCUDA provide low and high level abstraction for highly parallel computations on GPUs

Parallelism Best Practices

- Don’t cross the streams!
- Choose a form of parallelism – maybe two and stick to it! Trouble begins when you have:
  - multiple OpenMP runtimes or pthreads+OpenMP
  - **`multiprocessing`** (never the correct answer)
  - forking
- Watch affinity very carefully on the Cray – **`numpy`** and others can link threaded BLAS and LAPACK leading to more threads than you expect

Why MPI?

It is (still) the HPC paradigm for inter-process communications

- Supported by every HPC center and vendor on the planet
- APIs are stable, standardized, and portable across platforms and languages
- We'll still be using it in 10 years…

It makes full use of HPC interconnects and hardware

- Abstracts aspects of the network that may be very system specific
- Dask, Spark, Hadoop, and Protocol Buffers use sockets or files!
- Vendors generally optimize MPI for their hardware and software

Well-supported tools for development – even for Python

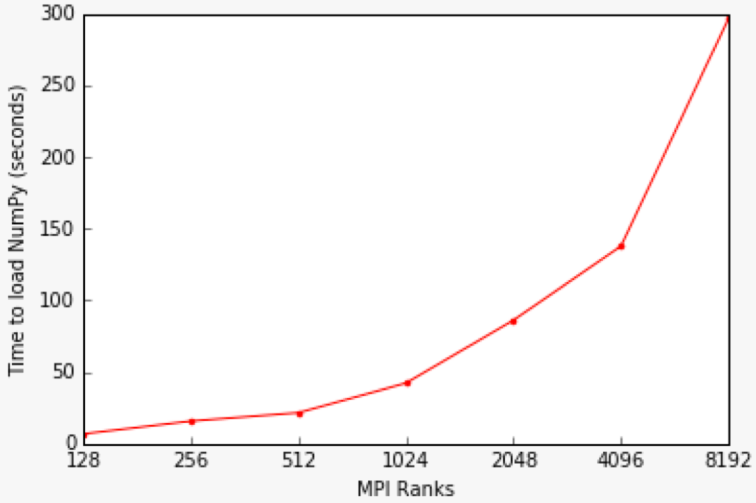
- Debuggers now handle mixed language applications
- Profilers are treating Python as a first-class citizen
- Many parallel solver packages have well-developed Python interfaces

Folks have been writing Python MPI bindings since at least 1996

- David Beazley may have started this…
- Other contenders: Pypar (Ole Nielsen), pyMPI (Patrick Miller, et al), Pydusa (Timothy H. Kaiser), and Boost MPI Python (Andreas Klöckner and Doug Gregor)
- The community has mostly settled on mpi4py by Lisandro Dalcin

A bottleneck at the start: Loading Python

When working in diskless environments or from shared file systems, keep track of how much time is spent in startup and module file loading. Parallel file systems are generally optimized for large, sequential reads and writes. NFS generally serializes metadata transactions. This load time can have substantial impact on total runtimes.



mpi4py

- Pythonic wrapping of the system's native MPI
- provides almost all MPI-1,2 and common MPI-3 features
- very well maintained
- distributed with major Python distributions
- portable and scalable
  - requires only: NumPy, Cython, and an MPI
  - used to run a python application on 786,432 cores
  - capabilities only limited by the system MPI
- http://mpi4py.readthedocs.io/en/stable/

How mpi4py works...

- mpi4py jobs are launched like other MPI binaries:
  `aprun –n ${RANKS} –N ${RANKS_PER_NODE} python ${PATH_TO_SCRIPT}`
- an independent Python interpreter launches per rank
  - no automatic shared memory, files, or state
  - crashing an interpreter does crash the MPI program
  - it is possible to embed an interpreter in a C/C++ program and launch an interpreter that way

How mpi4py works...

If you have trouble with simple MPI codes, remember:

- CPython is a C binary and mpi4py is a binding
- you will likely get core files and mangled stack traces
- use `ld` to check which MPI mpi4py is linked against – frequently a non-Cray MPI gets sucked in
- ensure Python, mpi4py, and your code are available on all nodes and libraries and paths are correct
  - on the Cray, it may be necessary to copy `${CRAY_MPICH_DIR}/lib/` into your environment's `${CONDA_PREFIX}/lib/`
  - Failure appears as there only being a single rank

mpi4py startup and shutdown

- Importing and MPI initialization
  - importing mpi4py allows you to set runtime configuration options (e.g. automatic initialization, thread_level) via `mpi4py.rc()`
  - by default importing the MPI submodule calls `MPI_Init()`
    - calling `Init()` or `Init_thread()` more than once violates the MPI standard
    - This will lead to a Python exception or an abort in C/C++
    - use `Is_initialized()` to test for initialization
- `MPI_Finalize()` will automatically run at interpreter exit
  - there is generally no need to ever call `Finalize()`
  - use `Is_finalized()` to test for finalization if uncertain
  - calling `Finalize()` more than once exits the interpreter with an error and may crash C/C++/Fortran modules

mpi4py and program structure

Any code, even if after `MPI.Init()`, unless reserved to a given rank will run on all ranks:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
mpisize = comm.Get_size()

if rank%2 == 0:
  print("Hello from an even rank: %d" %(rank))

comm.Barrier()

print("Goodbye from rank %d" %(rank))
```

mpi4py and datatypes

- Python objects, unless they conform to a C data type, are pickled
  - pickling and unpickling have significant compute overhead
  - overhead impacts both senders and receivers
  - pickling may also increase the memory size of an object
  - use the lowercase methods, eg: `recv(),send()`
- Picklable Python objects include:
  - `None, True, and False`
  - integers, long integers, floating point numbers, complex numbers
  - normal and Unicode strings
  - tuples, lists, sets, and dictionaries containing only picklable objects
  - functions defined at the top level of a module
  - built-in functions and classes defined at the top level of a module
  - instances of such classes whose `__dict__()` or the result of calling `__getstate__()` is picklable

mpi4py and datatypes

- Buffers, MPI datatypes, and NumPy objects aren't pickled
  - transmitted near the speed of C/C++
  - NumPy datatypes are autoconverted to MPI datatypes
  - buffers may need to be described as a 2/3-list/tuple
    `[data, MPI.DOUBLE]` for a single double
    `[data,count,MPI.INT]` for an array of integers
  - custom MPI datatypes are still possible
  - use the capitalized methods, e.g.: `Recv(), Send()`
- When in doubt: can it be represented as a memory buffer or only as `PyObject`?

mpi4py: collectives and operations

- Collectives operating on Python objects are naïve
- For the most part collective reduction operations on Python objects are serial
- Casing convention applies to methods:
  - **lowercase methods** will work for general **Python objects** (albeit slowly)
  - **uppercase methods** will work for **NumPy/MPI data types at near C speed**
  - **uppercase methods** will use **optimized vendor collectives**

mpi4py: Parallel I/O

- All 30-something MPI-2 methods are supported
- conventional Python I/O is not MPI safe!
  - safe to read files, though there might be locking issues
  - write a separate file per rank if you must use Python I/O
- h5py 2.2.0 and later support parallel I/O
  - hdf5 must be built with parallel support
    - make sure your hdf5 matches your MPI
    - **`h5pcc`** must be present
    - check things with: **`h5pcc -showconfig`**
    - hdf5 and h5py from Anaconda are serial!
  - anything which modifies the structure or metadata of a file must be done collectively
  - Generally as simple as:

```
f = h5py.File('parallel_test.hdf5', 'w',
              driver='mpio', comm=MPI.COMM_WORLD)
```

mpi4py: crashing

If you crash:

- Again, remember: CPython is a C binary and mpi4py is a binding
- You will likely get core files and mangled stack traces
- Use `ld` to check which MPI mpi4py is linked against
  - `mpi4py.get_config()` will show you the contents of `mpi.cfg` used at build time and is generally of limited utility
- Ensure Python, mpi4py, and your code are available on all nodes
- Ensure libraries and paths to files are correct in your scripts
- Try running with a single rank
- Rebuild binary modules with debugging symbols
- The default error handler is `MPI.ERRORS_RETURN` which allows the use of Python exception handling, but can allow for silent death in C/C++/Fortan MPI code.
  - Use `MPI.{Comm|Win|File}.Set_errhandler()` to set `MPI.ERRORS_ARE_FATAL` on any communicator, memory window, or file you pass into C/C++/Fortan MPI code.
  - Use `MPI.{Comm|Win|File}.Get_errhandler()` to check the error handler on any communicator, memory window, or file passed from C/C++/Fortan MPI code.

Hands-on 2

Notes:

- We have a Theta reservation for use in this training, the queue is `training`
- Examples assume exactly 8 nodes unless specified
- Ask questions if you get confused or something breaks

We'll create a simple submission script and run a few programs using mpi4py.

Hands-on Exercise 2: Using mpi4py

Instructions:

1. Check out the examples repo:
   `git clone https://github.com/wscullin/ecp_python_tutorial.git`

2. Change into the directory `ecp_python_tutorial`

3. Create a script submit.sh using your preferred environment

4. Submit `qsub –A yourproject ./submit.sh`

Hands-on Exercise 2: Using mpi4py (part 2)

Instructions:

5. Output should look like:

```
mpirun -n 8 $(which python) ./basic_features.py
##########################################################################
Rank 0 sees local_dict as {'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4, 'g': 'gee whiz', 'f': 6, 'h': ('hi', 'there')}
Rank 0 sees local_list_max as [0, 1, 2, 3]
Rank 0 sees local_list_sum as [0, 1, 2, 3]
Rank 0 sees local_string as This is a string.
Rank 0 sees local_tuple as (0, 0, 0, 0, 0, 0, 0, 0)
Rank 0 sees local_np_array as [0 1 2 3 4 5 6 7 8 9]
##########################################################################
Rank 6 sees local_dict as None
Rank 6 sees local_list_max as [0, 6, 12, 18]
Rank 6 sees local_list_sum as [0, 6, 12, 18]
Rank 6 sees local_string as This should be fun!
Rank 6 sees local_tuple as (6, 6, 6, 6, 6, 6, 6, 6)
Rank 6 sees local_np_array as [0 1 2 3 4 5 6 7 8 9]
##########################################################################

Running collective operations

##########################################################################
Rank 0 sees local_dict as a after scatter using None
Rank 0 sees local_list_max as [0, 7, 14, 21] after allreduce using max
Rank 0 sees local_list_sum as [0, 1, 2, 3, 0, 1, 2, 3, 0, 2, 4, 6, 0, 3, 6, 9, 0, 4, 8, 12, 0, 5, 10, 15, 0, 6, 12, 18, 0, 7, 14, 21] after reduce using sum
Rank 0 sees local_string as This is a string. after bcast using defaults
Rank 0 sees local_tuple as [0, 1, 2, 3, 4, 5, 6, 7] after alltoall using defaults
Rank 0 sees local_np_array as [ 0  8 16 24 32 40 48 56 64 72] after allreduce using sum
##########################################################################
Rank 6 sees local_dict as f after scatter using None
Rank 6 sees local_list_max as [0, 7, 14, 21] after allreduce using max
Rank 6 sees local_list_sum as None after reduce using sum
Rank 6 sees local_string as This is a string. after bcast using defaults
Rank 6 sees local_tuple as [0, 1, 2, 3, 4, 5, 6, 7] after alltoall using defaults
Rank 6 sees local_np_array as [ 0  8 16 24 32 40 48 56 64 72] after allreduce using sum
```

Hands-on Exercise 2: Using mpi4py

6. Why wasn't our output in order?
7. How might we scatter a dictionary?

8. Change to the directory pi
```
cd ecp_python_tutorial/pi
```

9. Run builtins_mpi_pi.py on 1, 8, and 16 ranks:
```
aprun -n 1 -N 1 python builtins_mpi_pi.py
aprun -n 8 -N 1 python builtins_mpi_pi.py
aprun -n 16 -N 2 python builtins_mpi_pi.py
```

10. Run threads_pi.py with 1, 8, and 16 threads with the same sample count:
```
./threads_pi.py 12000000 1
./threads_pi.py 12000000 8
./threads_pi.py 12000000 16
```
11. What does this tell us about native Python threads?

Enumerated admonishments

- Benchmark and profile as you develop
- Control your environment
- Ask if you can do an operation with NumPy or SciPy
- Watch your data types – use NumPy datatypes
- Never mix forking and threading – ie: Python multiprocessing
- Avoid threading in Python – use threads in compiled modules
- Check the build configurations of your important Python modules
- Beware of thread affinity:
  `aprun -n … -N … -e KMP_AFFINITY=none -d … -j …`
- Watch startup times carefully
- Search before you write code – someone else has likely already implemented the solution you seek
- On Cray systems, you'll need the `-b` flag to `aprun` with any sort of environment manager

Developing Your Own Bindings and Compiled Modules

While not an exhaustive, common options for using pre-compiled, vectorized, threaded, GIL-free code for speed from Python include:

Cython – create C code from Python or a Python-like language
F2PY – wrap Fortran code
PyBind11 – “seamless operability between C++11 and Python”
swig – generate bindings for just about anything
Boost.Python – “seamless operability between C++ and Python”
ctypes – built-in Python FFI for interfacing C **an option of last resort**
Writing bindings in C/C++ http://dan.iel.fm/posts/python-c-extensions/

Developing Your Own Modules: Cython

Cython is a meant to make writing C extensions easy
Naive usage can offer x12 speedups
Builds on Python syntax
Translates .pyx files to C which compiles
Provides interfaces for using functionality from OpenMP, CPython, libc, libc++, NumPy, and more
Works best when you can statically type variables
Lets you turn off the GIL
Provides annotations to guide development

Developing Your Own Modules: Cython

Using `cython -a ${sourcefile}.{pyx,py}`, we can get guidance on where a module built with Cython would have to interact with CPython and lose performance:

Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: calcpipy.c

```
+01: import random
 02:
+03: def calcpi_py(samples):
 04:     """serially calculate Pi using only standard library functions"""
+05:     inside = 0
+06:     random.seed(0)
+07:     for i in range(int(samples)):
+08:         x = random.random()
+09:         y = random.random()
+10:         if (x*x)+(y*y) < 1:
+11:             inside += 1
+12:     return (4.0 * inside)/samples
```

Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: calcpi.c

```
+01: cdef extern from "stdlib.h":
 02:     cpdef long random() nogil
 03:     cpdef void srandom(unsigned int) nogil
 04:     cpdef const long RAND_MAX
 05:
+06: cdef double randdbl() nogil:
 07:       cdef double r
+08:       r = random()
+09:       r = r/RAND_MAX
+10:       return r
 11:
+12: cpdef double calcpi(const int samples):
 13:     """serially calculate Pi using Cython library functions"""
 14:     cdef int inside, i
 15:     cdef double x, y
 16:
+17:     inside = 0
 18:
+19:     srandom(0)
+20:     for i in range(samples):
+21:         x = randdbl()
+22:         y = randdbl()
+23:         if (x*x)+(y*y) < 1:
+24:             inside += 1
+25:     return (4.0 * inside)/samples
 26:
```

Developing Your Own Modules: f2py

f2py comes with NumPy and can be used to rapidly generate wrappers for Fortran code

```
$cat calcpi.f90

    subroutine calcpi(samples, pi)
      REAL,INTENT(OUT) :: pi
      INTEGER, INTENT(IN) :: samples
      REAL :: x, y
      INTEGER :: i, inside

      inside = 0

      do i = 1, samples

        call random_number(x)
        call random_number(y)

        if ( x**2 + y**2 <= 1.0D+00 ) then
          inside = inside + 1
        end if

       end do
       pi = 4.0 * REAL(inside) / REAL(samples)
    end subroutine

$f2py --fcompiler=gfortran -m calcpi_fortran -c calcpi.f90
$...
$python -c "import calcpi_fortran; print calcpi_fortran.calcpi(1000000)"
3.14163589478
```

Other Tools for Performance

There are a handful of projects that seek to improve performance of pure Python code. Two noteworthy options are:

Numba – a Python JIT
- Sponsored by Continuum (now Anaconda, Inc.)
- Can target CPUs and GPUs
- Relies on decorators

PyPy – an alternative to CPython
- Not yet 100% compatible with CPython and all modules
- No code changes required

Hands-on Cross-Compiling on Cray XC40s with pip

```
virualenv --python=python2.7 "${VENV_NAME}"
source "${VENV_NAME}/bin/activate"

# If pip is badly out of date, the TLS certificates may not be trusted.
pip install --trusted-host pypi.python.org --upgrade pip

Set envvars needed to guide pip for cross-compiling and instruct it to build from source
CC=cc MPICC=cc pip install -v --no-binary :all: mpi4py

Set envvars needed for pip to use external dependencies. See package documentation.
HDF5_DIR="${CRAY_HDF5_DIR}/${PE_ENV}/${GNU_VERSION%.*}"
CC=cc HDF5_MPI="ON" HDF5_DIR="${HDF5_DIR}" pip install -v --no-binary :all: h5py
deactivate "${VENV_NAME}"
```

Questions?

See also:

ECP Python Tutorial:

https://github.com/wscullin/ecp_python_tutorial

by William Scullin (ALCF), Matt Belhorn (OLCF), and Rollin Thomas (NERSC)

Intel Python Distribution:

http://software.intel.com/en-us/distribution-for-python